21世纪高等学校规划教材 | 计算机科学与技术

中国大学MOOC“在线开放课程”配套教材

Java语言程序设计（MOOC版）

阚道宏　编著

清华大学出版社

21 世纪高等学校规划教材·计算机科学与技术
中国大学 MOOC“在线开放课程”配套教材

Java 语言程序设计
（MOOC 版）

阚道宏　编著

清华大学出版社
北　京

内 容 简 介

本书内容共分为 Java 基础语法、面向对象程序设计方法和 Java 应用程序开发 3 部分。本书不是简单重复 C 语言的学习过程来学习第二门编程语言，而是在 C 语言基础上的递进加强。学习完本书内容，读者将具备中级应用程序开发的能力。本书在讲解 Java 程序设计知识的同时系统介绍了相关的应用场景和背景知识，知识体系完整。本书在中国大学 MOOC（http://www.icourse163.org/）上同步开设配套的慕课（MOOC）课程，供读者免费学习。

开设“Java 语言程序设计”课程的教师可将本书作为授课教材使用，参加在线课程学习的学生可将本书作为线下阅读教材。

图书在版编目(CIP)数据

Java 语言程序设计：MOOC 版/阚道宏编著.—北京：清华大学出版社，2019
(21 世纪高等学校规划教材·计算机科学与技术)
ISBN 978-7-302-53017-6

Ⅰ.①J…　Ⅱ.①阚…　Ⅲ.①JAVA 语言－程序设计－高等学校－教材　Ⅳ.①TP312.8

中国版本图书馆 CIP 数据核字(2019)第 093903 号

责任编辑：付弘宇　张爱华
封面设计：傅瑞学
责任校对：焦丽丽
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 装 者：清华大学印刷厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：30.25　　**字　　数**：727 千字
版　　次：2019 年 9 月第 1 版　　**印　　次**：2019 年 9 月第1 次印刷
印　　数：1～1500
定　　价：69.00 元

产品编号：080377-01

出版说明

随着我国改革开放的进一步深化，高等教育也得到了快速发展，各地高校紧密结合地方经济建设发展需要，科学运用市场调节机制，加大了使用信息科学等现代科学技术提升、改造传统学科专业的投入力度，通过教育改革合理调整和配置了教育资源，优化了传统学科专业，积极为地方经济建设输送人才，为我国经济社会的快速、健康和可持续发展以及高等教育自身的改革发展做出了巨大贡献。但是，高等教育质量还需要进一步提高以适应经济社会发展的需要，不少高校的专业设置和结构不尽合理，教师队伍整体素质亟待提高，人才培养模式、教学内容和方法需要进一步转变，学生的实践能力和创新精神亟待加强。

教育部一直十分重视高等教育质量工作。2007 年 1 月，教育部下发了《关于实施高等学校本科教学质量与教学改革工程的意见》，计划实施"高等学校本科教学质量与教学改革工程"（简称"质量工程"），通过专业结构调整、课程教材建设、实践教学改革、教学团队建设等多项内容，进一步深化高等学校教学改革，提高人才培养的能力和水平，更好地满足经济社会发展对高素质人才的需要。在贯彻和落实教育部"质量工程"的过程中，各地高校发挥师资力量强、办学经验丰富、教学资源充裕等优势，对其特色专业及特色课程（群）加以规划、整理和总结，更新教学内容、改革课程体系，建设了一大批内容新、体系新、方法新、手段新的特色课程。在此基础上，经教育部相关教学指导委员会专家的指导和建议，清华大学出版社在多个领域精选各高校的特色课程，分别规划出版系列教材，以配合"质量工程"的实施，满足各高校教学质量和教学改革的需要。

为了深入贯彻落实教育部《关于加强高等学校本科教学工作，提高教学质量的若干意见》精神，紧密配合教育部已经启动的"高等学校教学质量与教学改革工程精品课程建设工作"，在有关专家、教授的倡议和有关部门的大力支持下，我们组织并成立了"清华大学出版社教材编审委员会"（以下简称"编委会"），旨在配合教育部制定精品课程教材的出版规划，讨论并实施精品课程教材的编写与出版工作。"编委会"成员皆来自全国各类高等学校教学与科研第一线的骨干教师，其中许多教师为各校相关院、系主管教学的院长或系主任。

按照教育部的要求，"编委会"一致认为，精品课程的建设工作从开始就要坚持高标准、严要求，处于一个比较高的起点上。精品课程教材应该能够反映各高校教学改革与课程建设的需要，要有特色风格、有创新性（新体系、新内容、新手段、新思路，教材的内容体系有较高的科学创新、技术创新和理念创新的含量）、先进性（对原有的学科体系有实质性的改革和发展，顺应并符合 21 世纪教学发展的规律，代表并引领课程发展的趋势和方向）、示范性（教材所体现的课程体系具有较广泛的辐射性和示范性）和一定的前瞻性。教材由个人申报或各校推荐（通过所在高校的"编委会"成员推荐），经"编委会"认真评审，最后由清华大学出版

社审定出版。

目前,针对计算机类和电子信息类相关专业成立了两个“编委会”,即“清华大学出版社计算机教材编审委员会”和“清华大学出版社电子信息教材编审委员会”。推出的特色精品教材包括:

(1) 21世纪高等学校规划教材·计算机应用——高等学校各类专业,特别是非计算机专业的计算机应用类教材。

(2) 21世纪高等学校规划教材·计算机科学与技术——高等学校计算机相关专业的教材。

(3) 21世纪高等学校规划教材·电子信息——高等学校电子信息相关专业的教材。

(4) 21世纪高等学校规划教材·软件工程——高等学校软件工程相关专业的教材。

(5) 21世纪高等学校规划教材·信息管理与信息系统。

(6) 21世纪高等学校规划教材·财经管理与应用。

(7) 21世纪高等学校规划教材·电子商务。

(8) 21世纪高等学校规划教材·物联网。

清华大学出版社经过三十多年的努力,在教材尤其是计算机和电子信息类专业教材出版方面树立了权威品牌,为我国的高等教育事业做出了重要贡献。清华版教材形成了技术准确、内容严谨的独特风格,这种风格将延续并反映在特色精品教材的建设中。

清华大学出版社教材编审委员会
联系人:魏江江
E-mail:weijj@tup.tsinghua.edu.cn

前　言

1. 关于本书

2006 年，我国开始在高等院校开展本科专业工程认证工作，其目的是更新教育观念，以产出为导向来重构课程体系，从根本上提升本科教学质量。中国《工程教育认证标准》(2015 版)明确提出本科培养目标，本科生应具备将工程知识用于解决复杂工程问题的能力。这就要求本科课程体系应互相衔接，形成层次，共同服务于专业培养目标。同时还需加强实践教学，提升学生的工程能力。

本书针对计算机本科专业工程认证，将程序设计能力培养划分成程序设计基础(初级)、应用程序开发(中级)和专业研究开发(高级)3 个层次，分别以 C/C++作为初级入门语言、Java 作为中级应用程序开发语言、Python 作为高级专业研究开发语言。这 3 个层次互相衔接，并在实践教学内容上逐层递进、加强，使得计算机专业本科生在毕业时就能具备较高的应用和研究开发能力。本书通过学习 Java 语言程序设计来培养学生中级应用程序开发能力。

2. 本书特色

1) 面向中级应用程序开发能力培养

本书不是简单重复 C 语言的学习过程来学习第二门编程语言，而是在 C 语言程序设计基础上的递进加强。本书将 Java 语言的学习重点放在面向对象程序设计方法和基于 Java 开源生态圈开发应用程序上，它们是 Java 语言的精髓。在学习完本书内容之后，读者将具备中级应用程序开发的能力。

2) 多种应用编程场景

本书设计多种不同的应用编程场景，在讲解 Java 程序设计知识的同时会先介绍相关的应用场景和背景知识。例如，很多读者在学习程序设计之前并没有学过计算机网络课程，不具备学习网络编程的基础，本书在讲解网络编程时，会先介绍计算机网络的基本原理及相关概念、术语，将程序员应当具备的网络知识提炼出来，以通俗易懂的方式呈现给读者。在掌握了这些网络知识之后，读者就可以无障碍地学习后续网络编程部分的内容了。

3) 同步慕课(MOOC)课程

本书在中国大学 MOOC(http://www.icourse163.org/)上同步开设配套的慕课(MOOC)课程，供读者免费学习。

3. 内容摘要

本书内容按章节顺序可分为 3 部分，分别是 Java 基础语法(第 1、2 章)，面向对象程序设计方法(第 3、4 章)和 Java 应用程序开发(第 5～10 章)。

第 1 章　认识 Java 语言。学习要点如下。

- 学习 Java 语言程序设计，重点学习 Java 生态圈和应用编程。
- Java 语言和 C/C++很相似，但 Java 生态圈流行开源文化，具有更多可用的类库。

- Java 语言具有自己的特点，其中最主要的特点是跨平台。
- 立即搭建 Java 开发环境(JDK＋Eclipse)，编写自己的第一个 Java 程序。

第 2 章　Java 语言基础。学习要点如下。

- Java 语言的基础语法大量借鉴了 C/C++语言。具有 C/C++语言基础的读者在学习 Java 语言时，只需重点了解它与 C/C++语言之间的区别。
- 本章应尽快熟悉 Java 语言编程环境。建议具有 C/C++语言基础的读者把之前学习过的 C/C++程序改用 Java 语言重写一遍。
- 通过对比可以知道，程序设计语言虽然语法不同，但设计思想是一致的。

第 3 章　面向对象程序设计之一。学习要点如下。

- 深入理解面向对象程序设计方法的基本原理和设计过程。
- 掌握 Java 语言中类与对象的语法规则。
- 理解引用数据类型与基本数据类型之间的区别。
- 掌握 Java 语言中与数组相关的语法。
- 掌握 Java 语言多文件结构的管理方法，重点理解包和子目录之间的对应关系。

第 4 章　面向对象程序设计之二。学习要点如下。

- 学会使用组合和继承的方法来定义新类，这样可以提高类代码的开发效率。
- 应从提高算法代码重用性的角度去理解对象的替换与多态机制。
- 熟练掌握接口的定义和实现方法，并充分理解接口与超类的区别。
- 熟练掌握匿名类和匿名方法的简写形式。

第 5 章　Java 基础类库。学习要点如下。

- 熟练掌握 Java API 说明文档的阅读方法。
- 学习 Java API 的使用，例如数学类 Math、字符串类 String、基本数据类型的包装类、根类 Object 和系统类 System 等。
- 理解并掌握 Java 语言的 try-catch 异常处理机制。
- 理解泛型编程，并能通过 Java API 中的数据集合类实现动态数组、队列、堆栈、集合和映射等功能。
- 掌握 Java 语言文档注释和注解的基本用法。

第 6 章　图形用户界面程序。学习要点如下。

- 了解 Java API 中各图形组件之间的关系。
  - ◇ 框架窗口 JFrame 和对话框窗口 JDialog 是顶层容器，其中包含内容面板。
  - ◇ 可以在内容面板中添加组件，并可设置不同的布局管理策略。
  - ◇ 内容面板可使用 JPanel 划分出子面板，子面板独立布局，可实现比较复杂的图形界面。
- 了解 Java 图形用户界面程序的事件响应机制。
- 通过编程练习掌握常用组件的用法，并能根据程序功能要求设计图形用户界面。
- 在掌握上述图形用户界面基本编程原理之后，可通过 Java API 文档自行研究 javax.swing 包中其他各种不同功能的图形组件，例如 JSplitPane、JTabbedPane、JEditorPane、JPasswordField、JPopupMenu、JToolBar、JToolTip、JProgressBar、JScrollBar、JSlider、JSpinner、JTree 等。

第 7 章　输入输出流。学习要点如下。

- Java API 中的类往往经历了多级抽象和多层包装，例如输入输出流类族中的类。读者在学习 Java API 过程中要注意及时总结并梳理出类与类之间的继承或包装关系。
- 初学者可以从常用类开始，先学习使用，然后再追溯其超类。逐步从微观到宏观，最终实现从整体上把握 Java API 类库的目标。
- 学习并掌握标准 I/O、文件 I/O 的常规编程方法和代码框架。
- 学习并掌握基本的文本处理方法，并能运用简单的正则表达式进行文本分析和处理。
- 学习并了解基本的图像及声音处理方法。

第 8 章　多线程并发编程。学习要点如下。

- 多线程是一种高级编程技术。多线程可以提高 CPU 使用率，改善用户体验。在多核或多 CPU 计算机系统上，使用多线程可以明显提高程序的运行速度。
- 要准确理解多线程编程中的 3 个要素。
  - ◇ 可以运行的算法对象，算法对象具有 run()方法。
  - ◇ 运行算法对象的线程对象，线程对象是 Thread 类的对象。
  - ◇ 被多个线程共享的数据对象，操作这些数据对象时需要启用同步（synchronized）机制，多线程协同还需要使用等待-唤醒（wait-notify）机制。
- 多线程编程比较复杂，学习时应仔细阅读并理解本章提供的示例程序，然后尝试自己重写一遍。

第 9 章　网络编程。学习要点如下。

- 了解计算机网络的基本原理，理解网络编程中常用的概念和术语。
- 学习并掌握基于 TCP 或 UDP 的网络应用程序代码框架，并能熟练运用 Java API 中相关的类进行网络编程。
- 掌握在单台计算机上调试网络应用程序的方法。
- 本章所学习的网络知识已基本能够满足网络编程的需要。如果希望深入学习计算机网络，读者可以进一步选修专门的计算机网络课程。

第 10 章　数据库编程。学习要点如下。

- 了解数据库的基本原理，学习 SQL 和 JDBC 编程框架。
- 熟练运用 JDBC API 编写数据库应用程序。
- 本章所学习的数据库知识已基本能够满足数据库编程的需要。如果希望深入学习数据库，读者可以选修专门的数据库课程，系统学习数据库相关的基础理论和设计方法。
- 开启自己的 Java 探索之旅。不忘初心，砥砺前行！

4. 使用建议

开设“Java 语言程序设计”课程的教师可将本书作为授课教材使用，联系作者可免费获得配套教学课件。参加在线课程学习的学生可将本书作为线下阅读教材。因水平所限，书中难免存在疏漏之处。如果您发现相关内容，烦请发送邮件告知作者，不胜感激。

如将本书作为课堂教学用书，则建议讲课学时和实验学时各为 32 学时，合计 64 学时。

每学时50分钟。作者按如下方式安排讲课学时：第1、2、8、9、10章各2学时，第3、4、6、7章各4学时，第5章6学时。

作者联系方式：kandaohong@cau.edu.cn。

5. 致谢

作者通过中国大学MOOC积累了一些在线课程教学的经验，所开设的“C++语言程序设计”被教育部认定为第一批“国家精品在线开放课程”。本书将继续在中国大学MOOC上同步开设配套在线课程“Java语言程序设计”，供读者免费学习。感谢中国大学MOOC！

本书的出版得到了清华大学出版社付弘宇编辑和张爱华编辑的热情帮助和悉心指导，这使得本书的文字质量得到了很大提升。在此表示衷心的感谢！

最后，感谢家人对我的理解和支持。

作　者

2019年3月于北京



第1部分 Java基础语法

第 2 部分 面向对象程序设计方法

第3部分 Java应用程序开发

第1部分 Java基础语法

第1章 认识Java语言

计算机程序(program)是使用某种计算机语言编写的一组指示计算机进行数据处理的指令序列(或称语句序列)。使用计算机处理数据一般可分为4个步骤。

(1) **申请内存空间**。数据要存放在内存中才能被CPU读取和处理,处理后的结果也只能保存回内存中。程序需要通过定义变量指令,预先为数据分配好内存单元。数据包括原始数据、中间结果和最终结果等。

(2) **输入原始数据**。计算机通过输入设备输入原始数据。程序通过输入指令将原始数据输入到预先分配好的内存单元中等待处理。键盘是最常用的输入设备,可以输入数值、文字等数据。

(3) **数据处理**。CPU负责数据处理。它从内存中读取原始数据,将处理结果再放回内存中。程序通过由不同运算符构成的表达式来对数据进行处理。

(4) **输出处理结果**。数据处理结束后,应当将处理结果通过输出设备反馈给用户。程序通过输出指令将存放在内存中的处理结果送往输出设备。显示器是最常用的输出设备,可以显示数值、文字、图形、图像等数据。

设计和编写计算机程序的人员称为程序员(programmer)。程序员通常使用高级语言编写程序,常用的高级语言有C、C++、Java、Python和C#等。高级语言程序中的一条指令通常被称为是一条语句(statement)。

目前,使用Java语言开发计算机程序的程序员比例最高,主要用于开发网络应用程序和安卓(Android)系统手机的App程序等。

1.1 从C/C++到Java

学习Java语言,最好具备C语言或C++语言基础。

1.1.1 Java语言与C/C++语言比较

本节通过一个具体的温度换算程序来比较Java语言与C/C++语言存在哪些异同。将摄氏温度换算成华氏温度的公式是:$f=c\times 1.8+32$,其中f表示华氏温度,c表示摄氏温度。

例1-1～例1-3分别给出了用C语言、C++语言和Java语言编写的温度换算程序。

例 1-1 一个用 C 语言编写的温度换算程序

```
 1  /*
 2    一个 C 程序实例:
 3    将摄氏温度换算成华氏温度.
 4   */
 5  #include <stdio.h>                  //插入头文件 stdio.h
 6
 7  int main()                          //主函数
 8  {
 9      double ctemp, ftemp;            //定义保存温度数据的变量
10      scanf( "%lf", &ctemp );         //输入摄氏温度
11      ftemp = ctemp * 1.8 + 32;       //计算华氏温度
12      printf( "%lf\n", ftemp );       //输出华氏温度
13      return 0;
14  }
```

请读者注意这样一个细节：例 1-1 的 C 语言程序通过调用**系统函数** scanf()、printf()实现了数据的输入(第 10 行)和输出(第 12 行)功能。为了使用这两个系统函数,程序第 5 行通过#include 指令插入了**头文件** stdio.h,因为使用系统函数必须“先声明,再调用”。

例 1-2 一个用 C++语言编写的温度换算程序

```
 1  /*
 2    一个 C++程序实例:
 3    将摄氏温度换算成华氏温度.
 4   */
 5  #include <iostream>                //插入头文件 iostream
 6  using namespace std;               //声明命名空间 std
 7
 8  int main()                         //主函数
 9  {
10      double ctemp, ftemp;           //定义保存温度数据的变量
11      cin >> ctemp;                  //输入摄氏温度
12      ftemp = ctemp * 1.8 + 32;      //计算华氏温度
13      cout << ftemp;                 //输出华氏温度
14      return 0;
15  }
```

例 1-2 的 C++程序改用输入输出流类的**对象** cin、cout 实现了数据的输入(第 11 行)和输出(第 13 行)功能。

例 1-3 一个用 Java 语言编写的温度换算程序

```
 1  /*
 2    一个 Java 程序实例:
 3    将摄氏温度换算成华氏温度.
 4   */
 5  import java.util.Scanner;                      //导入外部程序 Scanner
 6
 7  public class JavaTemp {                        //先定义一个类
```

```
8       public static void main( String args[ ] ) {    //将主函数定义在类的里面
9           double ctemp, ftemp;                      //定义保存温度数据的变量
10          Scanner sc = new Scanner( System.in );    //创建键盘扫描器对象
11          ctemp = sc.nextDouble();                  //输入摄氏温度
12          ftemp = ctemp * 1.8 + 32;                 //计算华氏温度
13          System.out.println( ftemp );              //输出华氏温度
14          return;
15      }
16  }
```

在 Java 语言中，输入输出流类的**对象**是 System.in 和 System.out。例 1-3 的 Java 程序分别使用这两个对象来实现输入(第 10～11 行)和输出(第 13 行)的功能。另外，Java 语言习惯上将左大括号“{”放在上一行语句的后面，例如例 1-3 的第 8 行。

1. Java 语言与 C/C++语言的相似之处

Java 语言借鉴了 C/C++语言的语法，具有类似的语言风格。Java 程序与 C/C++程序的相似之处主要体现在以下几个方面。

- 都有一个名为 main()的**主函数**，但 Java 程序的主函数需要被定义在类的里面。Java 语言规定，所有函数都必须定义在某个类中。例如例 1-3，Java 程序的主函数 main()就被定义在了类 JavaTemp 中。
- 定义变量的语法格式相同，**数据类型**也类似。例如，double 都表示双精度实数(浮点)类型。
- 用于数据处理的**运算符**和**表达式**语法相同。例如，+、-、*、/这 4 个运算符在 C/C++/Java 语言中都表示的是加、减、乘、除运算。
- 语句都以**分号**“;”结束。
- 程序**注释**的形式也类似。例如，“/* …… */”都用于表示多行注释，“//……”都用于表示单行注释。

2. Java 语言与 C/C++语言的不同之处

Java 程序与 C/C++程序的不同之处主要体现在输入和输出语句上。

- C 语言通过系统函数实现输入输出功能，例如调用函数 scanf()、printf()就可以进行数据的输入或输出。C 语言是**结构化程序设计**语言，**函数**是结构化程序设计中最典型的语法表现形式。
- C++语言全盘继承了 C 语言的语法，同时又增加了**面向对象程序设计**的语法。例如 C++程序可以调用函数 scanf()、printf()来进行输入输出，同时也可以使用输入输出流类的对象 cin、cout 来实现数据的输入或输出。C++语言既支持结构化程序设计方法，也支持面向对象程序设计方法。**类**与**对象**是面向对象程序设计中最典型的语法表现形式。
- Java 语言只支持面向对象程序设计方法，是一种“**纯**”面向对象的程序设计语言。Java 程序只能通过输入输出流类的对象才能完成数据的输入或输出。System.in 是一个输入流类的对象，可实现键盘输入的功能；System.out 是一个输出流类的对

象,可实现显示器输出的功能。

输入输出函数或输入输出流类的对象,实际上是计算机语言为程序员提供的程序零件。直接使用这些零件,程序员就可以轻松实现相应的程序功能。结构化程序设计以函数的语法形式来提供程序零件,而面向对象程序设计是以类和对象的语法形式来提供程序零件。

1.1.2 简单 Java 程序的代码框架

例 1-4 给出一个简单 Java 程序的代码框架。

例 1-4 简单 Java 程序的代码框架

```
1  import java.util.Scanner;            //导入外部程序 Scanner
2
3  public class 类名 {                   //先定义一个类,类名需与源程序文件名一致
4                                        //假设类名为"P1",则源程序文件名必须为"P1.java"
5      public static void main( String args[ ] ) {     //将主函数定义在类的里面
6          int x; double y;                             //定义变量,申请内存
7          Scanner sc = new Scanner( System.in );       //创建键盘扫描器对象
8          x = sc.nextInt(); y = sc.nextDouble();       //使用扫描器输入原始数据
9          …;                                           //编写表达式进行数据处理
10         System.out.println( … );                     //输出计算结果
11     }
12 }
```

1. Java 语言需要将主函数 main()定义在某个类中

包含主函数 main()的类被称为**主类**。保存主类代码的源程序文件名必须与类名一致。假设主类名为 P1,则其源程序文件名必须为 P1.java。

2. Java 语言的键盘输入

(1) 首先导入外部程序 Scanner(扫描器)。

```
import java.util.Scanner;
```

(2) 然后使用扫描器 Scanner 创建键盘扫描器对象。

```
Scanner sc = new Scanner( System.in );
```

其中:new Scanner(…)表示创建一个扫描器对象;System.in 表示键盘;sc 是一个扫描器类型的引用变量,指向所创建出的键盘扫描器对象。

(3) 使用键盘扫描器对象输入数据,可以输入不同类型的数据。例如:

```
int x = sc.nextInt();          //为 int 型变量输入数据
double x = sc.nextDouble();    //为 double 型变量输入数据
float x = sc.nextFloat();      //为 float 型变量输入数据
char x = sc.nextChar();        //为 char 型变量输入数据
…
```

3. Java 语言的显示器输出

Java 语言使用输出流对象 System.out 来表示显示器，其中包含两个常用方法（即函数）：print()和 println()。程序员使用这两个方法就可以在显示器上显示（即输出）数据或其他提示信息。使用这两个方法的语法形式如下：

```
System.out.print( … );                    //显示内容(不换行)
System.out.println( … );                  //显示内容后换一行
```

例如：

```
System.out.print( "Hello, world" );          //显示信息"Hello, world",显示后不换行
System.out.println( "Hello, world" );        //显示信息"Hello, world",显示后换一行
System.out.print( "Hello" + ", world\n" );   //显示信息"Hello, world",显示后换一行
System.out.println( "Hello, world" + 5 );    //显示结果: Hello, world5
int x = 5; double y = 10.6;
System.out.println( x + ", " + y );          //显示结果: 5, 10.6
```

1.1.3　如何学习程序设计

程序设计能力培养大致可分为**程序设计基础**（初级）、**应用程序开发**（中级）和**专业研究开发**（高级）3 个层次，如图 1-1 所示。这 3 个层次在学习内容上互相衔接，逐层递进、加强，最终让学习者达到较高的程序设计水平。

1. 程序设计基础（初级）

初级阶段的目标是学习程序设计的**基本原理**，其中包括计算机硬件结构及其工作原理，程序如何管理内存来存储数据（例如变量的定义与访问、数据类型、引用与指针等），程序如何指示 CPU 来处理数据（例如各种不同的运算符），或通过控制语句来控制指令的执行顺序等。

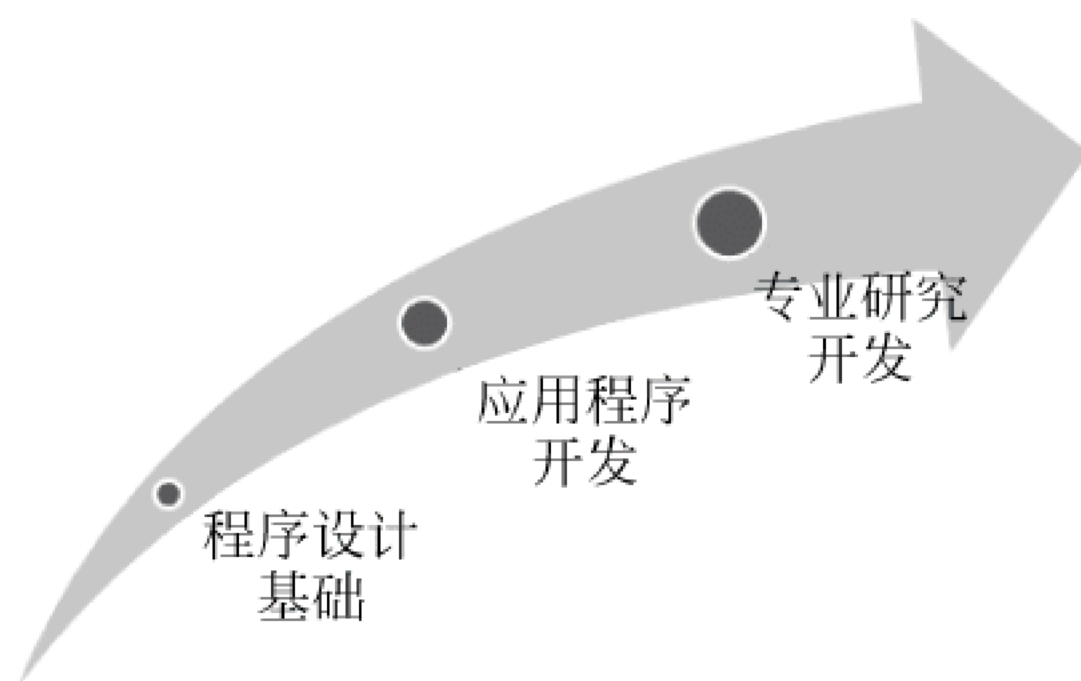


图 1-1　程序设计能力培养

初级阶段还应了解如何设计大型、复杂的程序，这就需要学习**程序设计方法**。程序设计方法有两种，分别是结构化程序设计和面向对象程序设计。

初级阶段学习结束后，读者可以参加计算机等级考试（二级）或各种程序设计大赛等活动，并在应试过程中进一步提高自己的水平。

2. 应用程序开发（中级）

中级阶段的目标是学习**应用程序开发**（学会就可能有好工作了哦）。应用程序开发需要基于其他人的程序零件来开发，从零开始是不可能的。

结构化程序设计方法规定：其他人给你**函数库**，你要会调用其他人的函数。**面向对象程序设计方法**规定：其他人给你**类库**，你要知道如何使用其他人的类库。因此在学习应用程序开发之前必须掌握结构化程序设计方法或面向对象程序设计方法。目前面向对象程序设计是主流，已经很少有人继续给程序员提供函数库了，所提供的大多是类库。

掌握了面向对象程序设计方法，只要有微软公司的类库（VC6.0 或 Visual Studio 提供），就可以使用 C++语言开发 Windows 程序；或者有谷歌公司的类库，就可以使用 Java 语言开发安卓（Android）智能手机的 App 了。

不同操作系统是由不同厂家开发的，它们对计算机语言的支持程度有所不同。例如，Windows 操作系统是由微软公司开发的，开发 Windows 软件主要使用 C++或 C# 语言；Mac OS/iOS 操作系统是由苹果公司开发的，开发 Mac OS/iOS 软件主要使用 Objective-C（C++的变种）或 Swift 语言；Android 操作系统由谷歌公司主导，开发 Android 软件主要使用 Java 语言。

开发网络应用程序主要使用 Java、C# 或 Python 语言；开发嵌入式应用程序主要使用 C、C++或 Java 语言；向量或矩阵计算、数据分析或人工智能研究常使用 Python 语言。

到底选择哪种计算机语言来开发应用程序，这不取决于程序员的主观意愿，而是取决于应用程序将来会在哪个操作系统上使用，或所开发应用程序的用途。

3. 专业研究开发（高级）

计算机专业的同学在学习完程序设计之后，还会进一步学习计算机领域的专业课程，例如计算机组成原理、数据结构、操作系统、编译原理、数据库原理、计算机网络、计算机图形学、数字图像处理、离散数学、算法设计、大数据、人工智能等。在学习专业课程或开展科学研究的过程中，可能会用到不同的计算机语言。例如，学习人工智能可能需要用到 Python 语言。这就需要计算机专业的同学具备自学新语言的能力，或者通过**网络在线开放课程**（例如 MOOC 课程）进行学习。

目前，越来越多非计算机专业的读者也开始学习程序设计。如果仅仅是希望了解程序设计，或者希望借助计算机程序开展本专业的科学研究，建议直接学习 Python 语言。因为 Python 语言有很多现成的程序零件库，易于上手。如果是希望进入软件行业，成为专业软件开发人员，还应当从 C、C++和 Java 语言开始学起。

1.1.4 Java 语言简介

围绕某一种计算机语言，有很多厂家或个人为其提供了编写好的函数库或类库，这就构成了一个以计算机语言为核心的**生态圈**。不同计算机语言具有不同的生态圈，参见图 1-2。

学习程序设计，通常以 C 或 C++语言作为零基础入门语言，主要学习程序设计原理和程序设计方法（包括结构化程序设计方法和面向对象程序设计方法）；然后再学习一门应用程序开发语言，例如 Java、C# 或 Python。和 C、C++或 C# 语言相比，Java 和 Python 语言具有**开源**（open source）的文化传统，有更多的程序零件库可供选择。

1. Java 语言的特点

- 借鉴了 C/C++语言的优良特性，具有相似的语法风格。
- 所有变量和函数代码都被定义在称为 **class** 的类中，程序中没有游离在类外的全局变量或外部函数，因此 Java 语言被称为是一个“纯”面向对象的计算机语言。
- 借助**字节码**（bytecode）和 Java **虚拟机**（Java Virtual Machine，**JVM**）实现一次编译，跨平台运行。

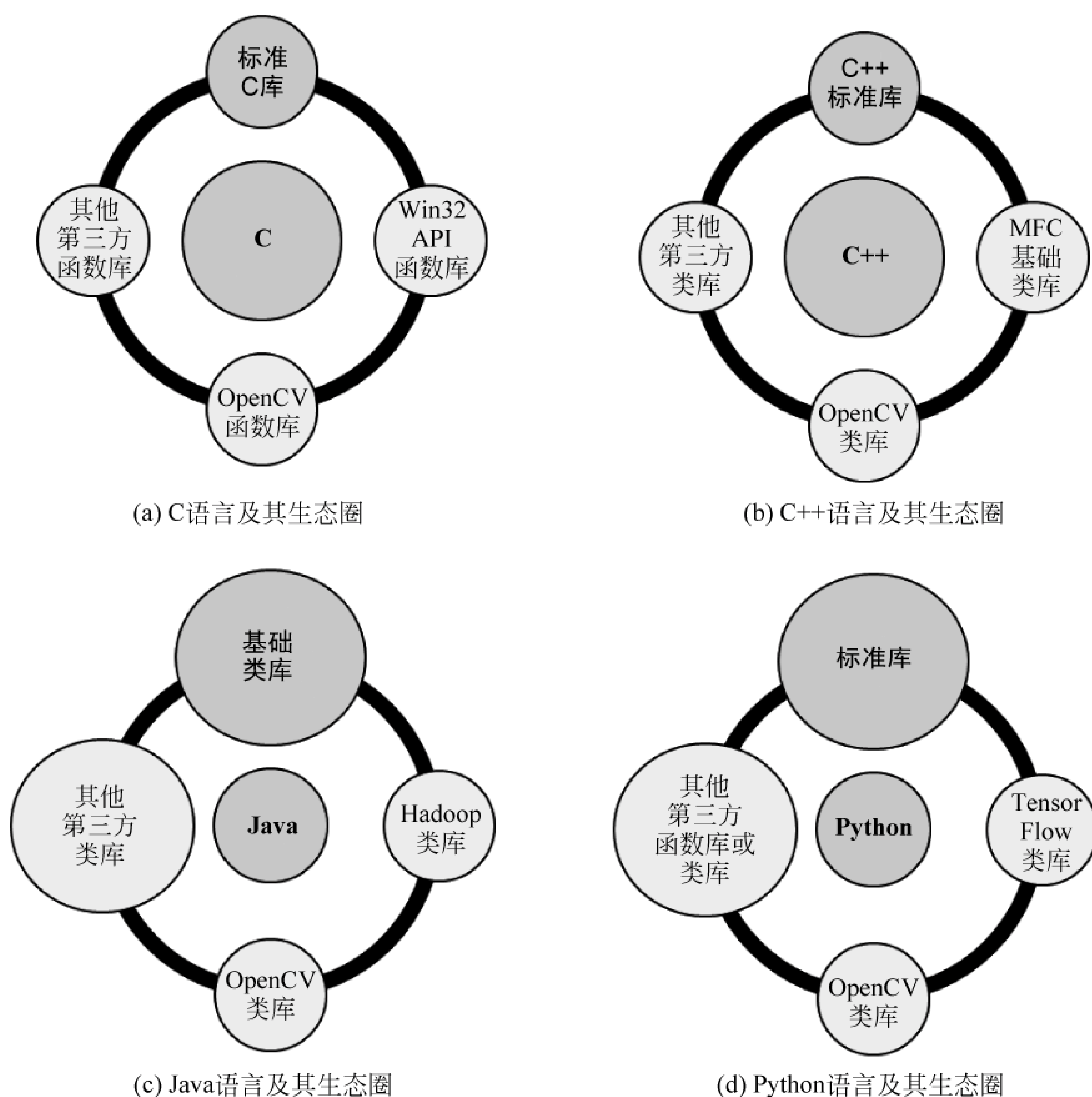


图 1-2 不同计算机语言及其生态圈比较

- 具有非常丰富的、可实现不同程序功能的类库，这些类库构成了以 Java 语言为核心的软件开发生态圈。
- 开源文化，免费共享。

学习 Java 语言，最好具备 C 语言或 C++语言基础。Java 语言与 C/C++语言有很多相似之处，因此可以快速学习 Java 语言的基础语法部分，然后将学习重点放在以下两个部分。

- 学习面向对象程序设计方法。
- 进入 Java 语言生态圈，学习程序的应用开发。重点是了解程序的**应用场景**，学会利用生态圈中已有的类库来快速开发程序。

2. Java 语言的发展历史

1995 年，Java 语言由美国 Sun 公司设计推出。2009 年，Oracle 公司收购了 Sun 公司，Java 语言转由 Oracle 公司进行升级维护，继续推出新版本。图 1-3 简要给出了 Java 语言的发展历史。

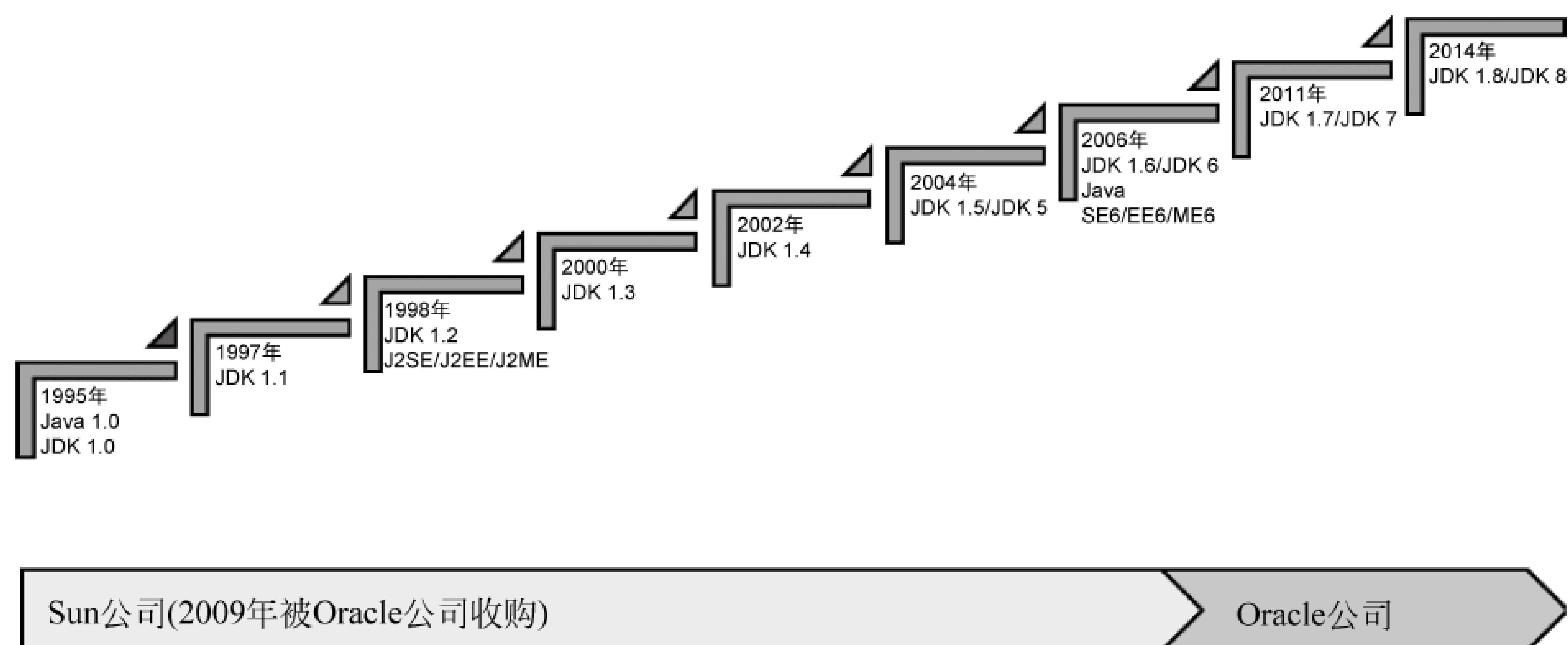


图 1-3 Java 语言的发展历史

本节习题

1. 使用计算机处理数据,输入原始数据的操作必须放在下列步骤(　　)的后面。
   A. 申请内存空间　B. 数据处理　C. 输出处理结果　D. 以上都不是
2. Java 语言与 C/C++语言在(　　)方面存在明显区别。
   A. 数据类型　B. 运算符　C. 表达式　D. 输入输出
3. 如果程序中出现关键字 class,则这个程序应该不会是用(　　)编写的。
   A. C 语言　B. C++语言　C. Java 语言　D. C# 语言
4. 下列写法中,(　　)是 Java 语言主函数 main()的正确写法。
   A. int main()　B. public static int main()
   C. public static void main(String args[])　D. public static int main(String args[])
5. Java 程序向显示器输出信息"Hello, World",(　　)的写法是错误的。
   A. System.out.print("Hello, world");
   B. System.out.println("Hello, world");
   C. System.out.print("Hello, world\n");
   D. printf("Hello, world\n");

1.2 Java 开发包 JDK

使用 Java 语言开发程序,需要用到 **Java 开发包**(Java Development Kit,**JDK**)。

1.2.1 JDK 的内容与版本

1. JDK 包含的内容

javac:Java 编译器。
java:Java 虚拟机。
javadoc:Java 文档生成器。

jar：Java 归档打包程序。

appletviewer：Java 小应用程序查看器。

Java API（Application Programming Interface）：Java 应用编程接口，这是一组 Java 类库。

……

开发结束后，Java 程序只需要 Java 虚拟机和 Java API 类库就能运行，因此这两项合起来也被称为 **Java 运行环境**（Java Runtime Environment，**JRE**）。

2. JDK 的版本

Java 语言自 1995 年推出之后，一直在持续不断地升级。按照从低到高的顺序，JDK 版本号依次为 JDK 1.0～JDK 1.5（JDK 5）、JDK 5～JDK 8。2004 年，JDK 版本编号做了一次调整，将 JDK 1.5 改为 JDK 5，这种编号方式一直沿用到今天。之后所推出的 JDK 新版本依次为 JDK 6、JDK 7……截至 2018 年 12 月，最新 JDK 版本为 JDK 11。

Java 语言在推出新版本时，还会根据用途将 JDK 分成 3 个不同的系列，它们分别是 Java SE（标准版，用于开发 Java 应用程序）、Java EE（企业版，用于开发 Java Web 应用程序）和 Java ME（小微版，用于开发 Java 嵌入式应用程序）。

本书所使用的 JDK 版本是 JDK 8/Java SE 系列，简称 **Java SE 8**。学习 Java 语言程序设计，要求学习者在一开始就能搭建起 Java 开发环境。下面就以 Java SE 8 为例，具体讲解如何在自己的计算机上快速搭建 Java 开发环境。搭建 Java SE 8 开发环境，需分 4 步完成：下载 Java SE 系列 JDK 8 安装包、安装 JDK 8、设置环境变量，最后是验证安装。

1.2.2 下载 JDK

使用浏览器从以下网址下载 JDK 安装包：

http://www.oracle.com/technetwork/java/index.html（Oracle 官网下的 Java 首页，见图 1-4）或 http://java.sun.com（早期的 Java 官网，将被自动重定向到上面的 Java 首页）。

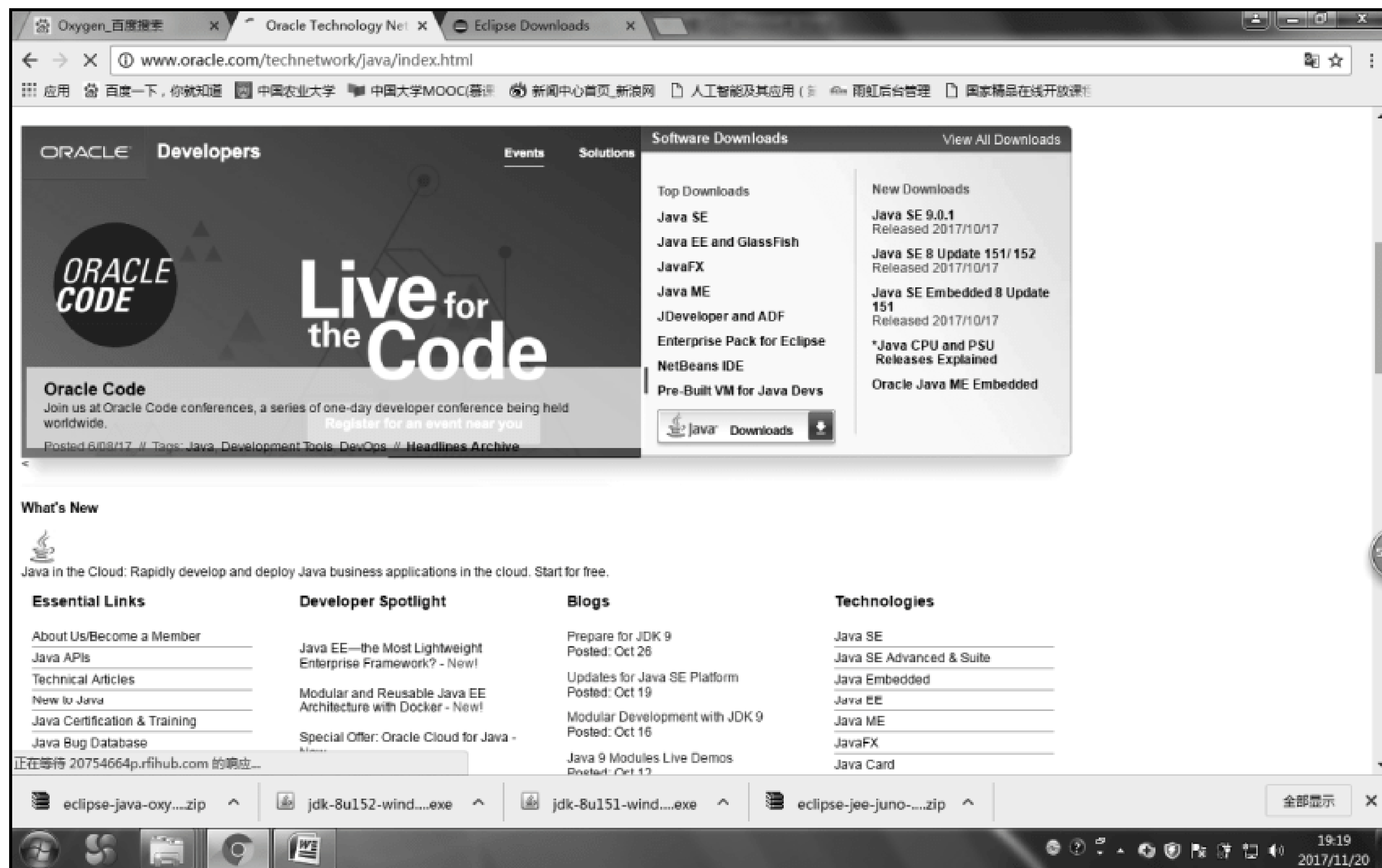


图 1-4 Oracle 公司官方网站下的 Java 首页

在图 1-4 中 Java 网页上部的 Top Downloads 中选择 Java SE,然后根据后续的链接指示下载 JDK 8 安装包。JDK 8 在正式推出后又进行了多次修订(update),形成不同的修订版本。另外,JDK 8 为不同操作系统制作了不同的安装包。本书下载的安装包是 jdk-8u152-windows-x64.exe,从这个安装包的名称可以解析出如下版本信息。

- 版本号,JDK8;修订版本号,u152。
- 这是为 Windows(64 位)操作系统制作的安装包。

读者可自行决定是否下载更新的 JDK 版本。

1.2.3 安装 JDK

JDK 安装包是一个可执行程序。运行安装包程序,进入"安装程序"界面(见图 1-5)。



图 1-5 安装包 jdk-8u152-windows-x64.exe 的"安装程序"界面

在图 1-5 中单击"**下一步**"按钮,进入"定制安装"界面(见图 1-6)。



图 1-6 "定制安装"界面

在图 1-6 中单击“**更改**”按钮，指定 JDK 的安装目录。本例将 JDK 安装到如下目录：

```
C:\Java\jdk1.8.0_152
```

然后单击“**下一步**”按钮，安装程序将开始复制文件，正式安装 JDK。JDK 安装结束后，安装程序还会继续安装运行环境 JRE。转至“目标文件夹”界面（见图 1-7），选择 JRE 安装目录，然后继续安装 JRE。**注：**后续将介绍的集成开发环境 Eclipse 会用到这个 JRE，因此必须安装。

本例在图 1-7 中单击“**更改**”按钮，将 JRE 安装到如下目录：

```
C:\Java\jre1.8.0_152
```

这是个新目录，需要单击“**新建文件夹**”按钮来创建这个目录（见图 1-8）。这样，JRE 就与 JDK 一起被安装到同一个根目录“C:\Java”中。



图 1-7　更改安装 JRE 的“目标文件夹”



图 1-8　浏览并新建 JRE 文件夹

在安装完 JDK 和 JRE 之后，安装程序就完成了全部安装任务。打开“C:\Java”文件夹，查看安装结果，如图 1-9 所示。

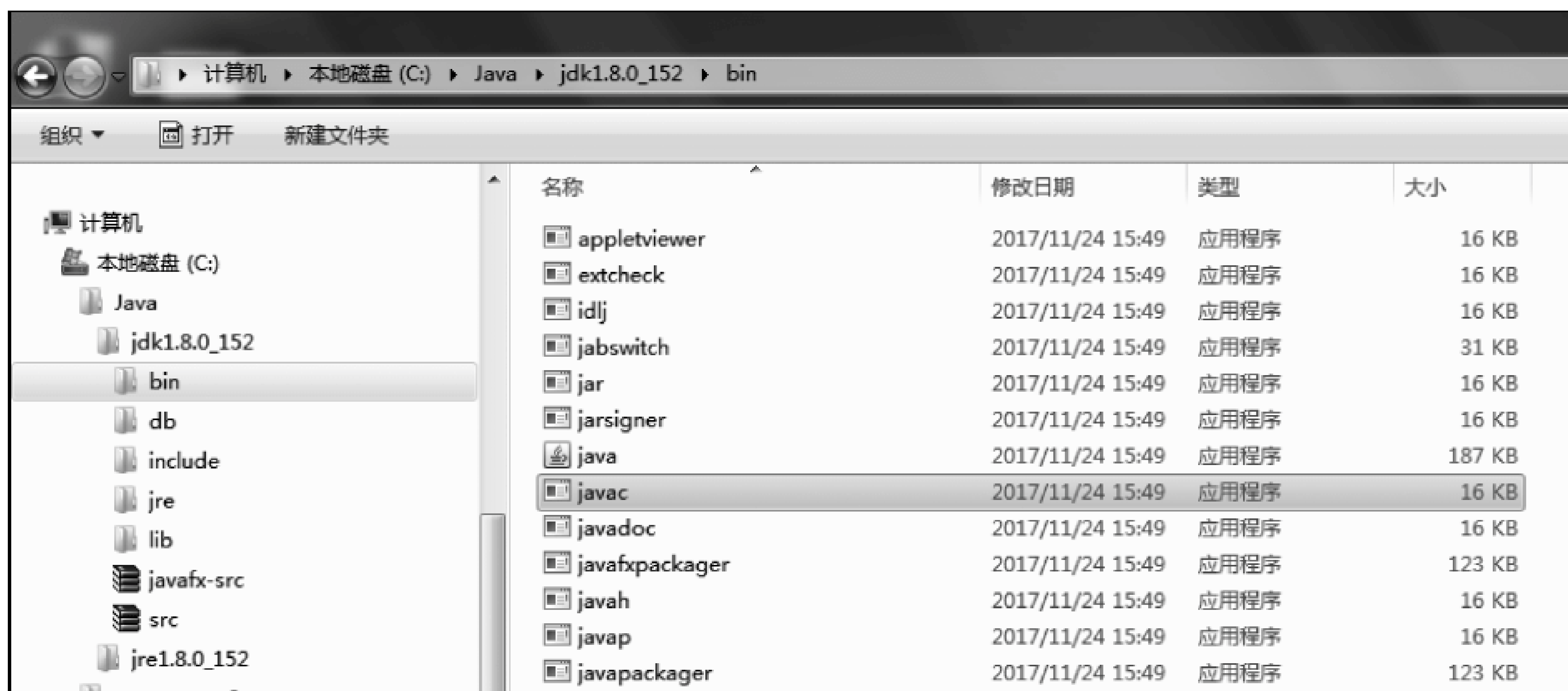


图 1-9　安装好的 JDK 和 JRE 目录结构

图 1-9 中的 javac(扩展名为.exe)就是 Java 语言的编译器,java(扩展名为.exe)则是执行 Java 程序的虚拟机 JVM(或称为 Java 解释器)。

1.2.4 设置 JDK

为便于使用,JDK 在安装完成之后还需要设置 3 个环境变量。这 3 个环境变量分别如下。

- **JAVA_HOME**:指明 JDK 的安装目录。
- **CLASSPATH**:指明查找 Java 类库时的搜索路径。
- **Path**(或 **PATH**):指明 Java 编译器及虚拟机等的安装目录。

在 Windows 操作系统上,需通过“**控制面板**”来设置环境变量。

1. 进入“控制面板”中的“环境变量”设置界面

启动“**控制面板**”,选择“**系统和安全**”→“**系统**”,进入图 1-10 所示的界面。



图 1-10 在“控制面板”中设置环境变量

在图 1-10 中单击左上角的“**高级系统设置**”,进入图 1-11 所示的“系统属性”设置界面。在图 1-11 中单击右下角的“**环境变量**”按钮,进入图 1-12 所示的“环境变量”设置界面。

2. 新建环境变量 JAVA_HOME

在图 1-12 中单击“**新建**”按钮,新建一个环境变量 **JAVA_HOME**,指明 JDK 的安装目录(见图 1-13)。

注:如果在“**用户变量**”中新建环境变量,则仅对该用户有效。如果希望对所有用户都有效,则应当选择在“**系统变量**”中新建环境变量。

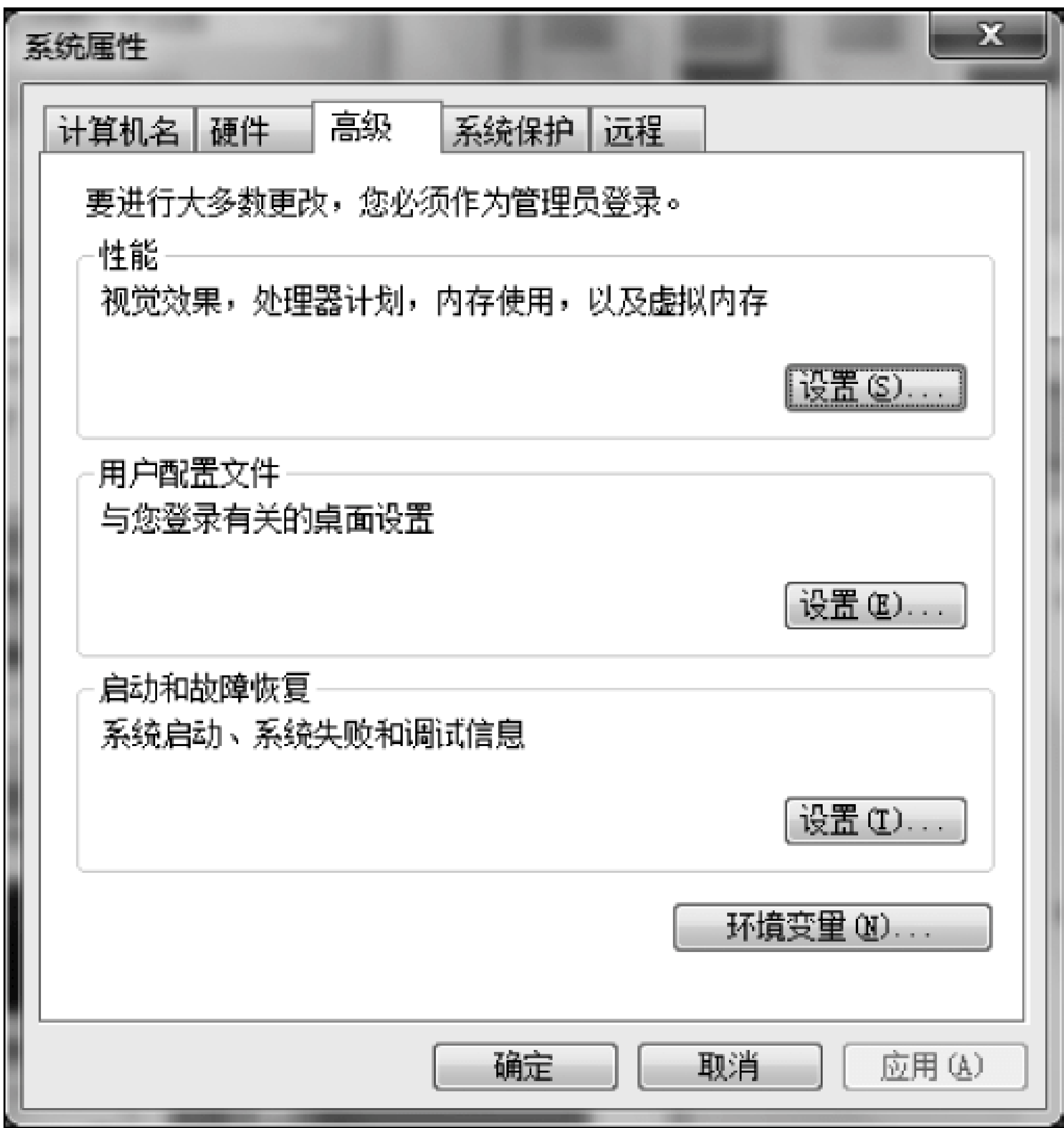


图 1-11　设置“系统属性”



图 1-12　设置“环境变量”

3. 新建环境变量 CLASSPATH

在图 1-12 中单击“**新建**”按钮，新建一个环境变量 **CLASSPATH**，指明查找 Java 类库时的搜索路径(见图 1-14)。



图 1-13　新建环境变量 JAVA_HOME

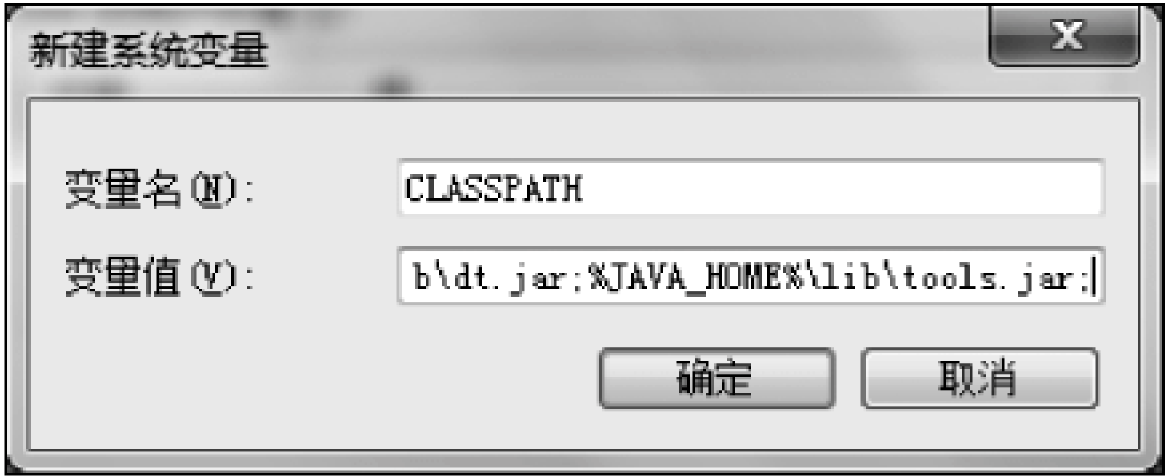


图 1-14　新建环境变量 CLASSPATH

应按如下格式来设置环境变量 CLASSPATH,其中包含 3 条由分号“;”(英文的半角分号)隔开的搜索路径：

```
.; %JAVA_HOME%\lib\dt.jar; %JAVA_HOME%\lib\tools.jar;
```

其中：第一个字符“.”表示当前目录,“%JAVA_HOME%”表示环境变量 JAVA_HOME 所设置的路径。各搜索路径之间用分号“;”隔开。

4. 设置环境变量 Path(或 PATH)

在图 1-12 的“**系统变量**”中查找到已有的环境变量 Path,然后单击“**编辑**”按钮弹出如图 1-15 所示的对话框。

在 Path 原有路径(**变量值**)的末尾添加 Java 编译器及虚拟机等的安装目录,例如：



图 1-15 编辑环境变量 Path

```
……; %JAVA_HOME%\bin;
```

注：“……”表示 Path 原有的路径。各路径之间用分号“;”隔开。

5. 验证 JDK 安装及其环境变量设置

搭建 Java SE 8 开发环境的最后一步是验证安装。进入 Windows 的命令行界面,验证 JDK 安装及其环境变量设置是否正确。在命令行界面中输入如下命令：

```
java  -version
```

如果运行结果如图 1-16 所示显示出了 Java 版本号,则说明 JDK 安装及其环境变量设置都是正确的。



图 1-16 Windows 命令行界面

本节习题

1. Java 开发包 JDK 中没有包含(　　)。
   A. Java 编译器　　B. Java 虚拟机
   C. Java 归档打包程序　　D. 头文件 stdio.h
2. 搭建 Java SE 8 开发环境需分 4 步,其中第 3 步是(　　)。
   A. 下载 JDK SE 8 安装包　　B. 安装 JDK SE 8
   C. 设置环境变量　　D. 验证安装
3. Java 运行环境 JRE 指的是(　　)。
   A. Java 编译器　　B. Java 虚拟机
   C. Java API　　D. Java 虚拟机+Java API
4. 使用 JDK 需要设置若干环境变量,其中不包括环境变量(　　)。
   A. JAVA_HOME　　B. CLASSPATH　　C. Path　　D. TEMP
5. 如果想在命令行界面中检查 JDK 版本,则应当输入命令(　　)。
   A. java -version　　B. cmd　　C. dir　　D. JDK -version

1.3 Java 程序和 Java 虚拟机

编写一个 Java 程序需分 3 步完成(图 1-17),它们分别是**编辑**、**编译**和**运行**。

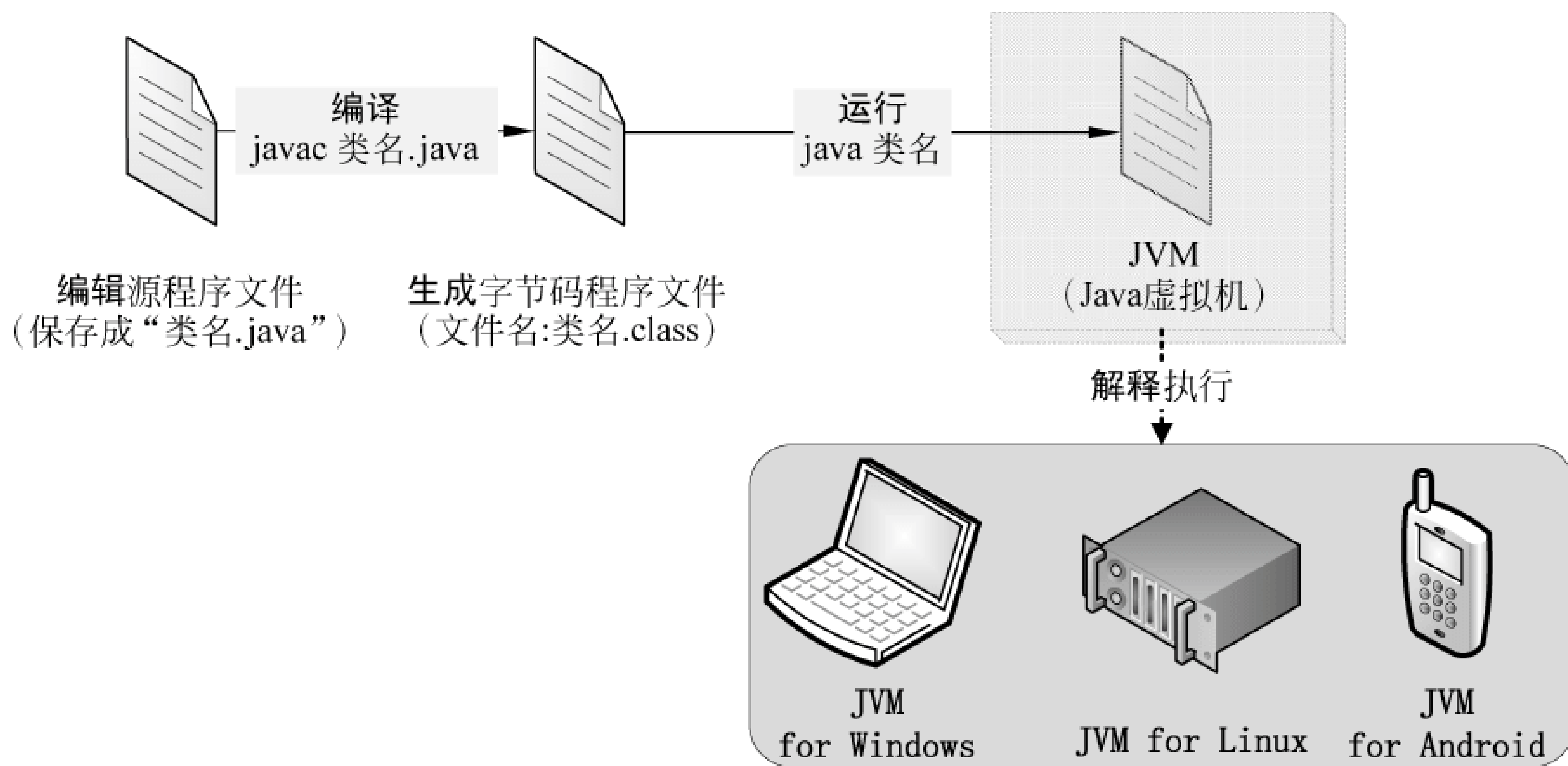


图 1-17　Java 程序的开发过程

1. 编辑

Java 语言以类的语法形式来编写程序代码。使用**文本型**编辑器软件输入编写好的 Java 类代码,并保存成硬盘文件。这个程序文件被称为 **Java 源程序**文件,其文件名应当与类同名,其扩展名为"**.java**"。

2. 编译

运行 Java **编译器**(\JDK 安装目录\bin**javac**.exe),可以将 Java 源程序文件(.java)编译成 Java **类程序**文件,其文件名也是类名,其扩展名为“**.class**”。Java 类程序文件相当于是一种机器语言程序,被称为字节码(bytecode)程序。

假设有一个已经编写好的 Java 类 test1,其源程序文件被保存在 d:\javatest 目录下的 test1.java 文件中。在控制台状态(例如 Windows 操作系统的 cmd 窗口),将源程序文件 test1.java 编译成类程序文件的过程分为如下两步。

(1) 首先将当前目录转到 d:\javatest。

(2) 然后运行 Java 编译器 javac.exe 对 test1.java 进行编译。

在 Windows 操作系统上,这两步操作所对应的控制台命令分别是(下面的回车键即 Enter 键):

```
cd  d:\javatest <回车键>
javac  test1.java <回车键>
```

Java 编译器对源程序文件 test1.java 进行编译,并将编译所生成的字节码程序保存成一个同名的类程序文件 test1.class。这两个程序文件的扩展名不同,但文件名是一样的,都是类名。

3. 运行

Java **虚拟机**负责加载、执行类程序文件中的字节码程序。Java 虚拟机实际上也是一个程序(\JDK 安装目录\bin**java**.exe),它模拟实现了用一个虚拟 CPU 执行程序的功能。例如,在 Windows 控制台状态下运行类程序文件 d:\javatest\test1.class 的命令为:

```
java  d:\javatest\test1 <回车键>
```

注 1:其中 test1 指定的是类名,不是文件名,即 test1 后面不能带.class。Java 虚拟机是根据类名去查找对应的 class 文件的。

注 2:只有包含主方法 main()的 Java 类才能执行。例如,要想执行类 test1 的程序代码,其中必须定义有主方法 main()。

Java 虚拟机有不同版本,可运行于不同**操作系统及真实 CPU** 之上。借助 Java 虚拟机,Java 程序无须重新编译即可在不同的计算机系统上运行,实现了跨平台运行。Java 虚拟机用软件模拟实现了一个虚拟 CPU,它和真实 CPU 一样定义了指令集、寄存器、字节码程序文件结构等。这个虚拟 CPU 的指令是一种介于高级语言和机器语言之间的“**中间语言**”。

Java 编译器将 Java 源程序中的 Java 语言代码编译成虚拟 CPU 的中间语言代码,这就是字节码。Java 虚拟机是以**解释方式**执行字节码程序的,其执行过程是:逐条取出字节码程序中的虚拟 CPU 指令,将其转换成真实 CPU 指令并在真实 CPU 上执行,边转换边执行。Java 虚拟机有时也被称作 **Java 解释器**。

Java 虚拟机执行字节码程序文件分为如下 3 步。

(1) **加载字节码**:由类**加载器**(class loader)完成。

(2) **校验字节码**:由字节码**校验器**(bytecode verifier)完成。

(3) **执行字节码**：由运行时**解释器**(runtime interpreter)完成。

本节习题

1. 编写一个 Java 程序需分 3 步，其中不包含(　　)。
   A. 编辑　　B. 编译　　C. 连接　　D. 运行
2. Java 源程序文件的扩展名是(　　)。
   A. .java　　B. .class　　C. .obj　　D. .exe
3. Java 类程序文件的扩展名是(　　)。
   A. .java　　B. .class　　C. .obj　　D. .exe
4. Windows 操作系统中 Java 编译器程序的文件名是(　　)。
   A. javac.exe　　B. java.exe　　C. javac.class　　D. jar.exe
5. 用(　　)编写的程序可以"一次编译，跨平台运行"。
   A. C 语言　　B. C++语言
   C. Java 语言　　D. 以上都不可以

1.4　Java 集成开发环境

程序员借助**集成开发环境**(Integrated Development Environment，IDE)软件，可以方便地开发 Java 语言程序。本书推荐使用 **Eclipse** 集成开发环境。

1.4.1　Eclipse 集成开发环境

Eclipse 是一个非常流行的开源集成开发环境，由 Eclipse 基金会管理，可免费下载。Eclipse 以插件的形式支持多种语言的开发，例如 Java、C、C++ 和 Python 等。使用浏览器从以下网址下载 Eclipse 安装包：

http://www.eclipse.org/downloads/

1. Eclipse 版本

Eclipse 有不同的版本，并且对 JDK 有**最低版本**要求(见表 1-1)。如果安装了 JDK 8，则推荐使用 4.7 版 Eclipse(代号为 Oxygen)。

表 1-1　Eclipse 及其 JDK 版本要求

版本代号	版本号	发行日期	JDK 最低版本
Callisto	3.2	2006 年	JDK 1.4
Europa	3.3	2007 年	JDK 1.5
Ganymede	3.4	2008 年	JDK 1.5
Galileo	3.5	2009 年	JDK 1.5
Helios	3.6	2010 年	JDK 1.5
Indigo	3.7	2011 年	JDK 1.5

续表

版本代号	版本号	发行日期	JDK 最低版本
Juno	4.2	2012 年	JDK 1.5
Kepler	4.3	2013 年	JDK 6
Luna	4.4	2014 年	JDK 6
Mars	4.5	2015 年	JDK 7
Neon	4.6	2016 年	JDK 8
Oxygen	**4.7**	**2017 年**	**JDK 8**

2. 安装 Eclipse

本书下载了 4.7 版 Eclipse(代号为 Oxygen)安装包 eclipse-java-oxygen-1a-win32-x86_64.zip,用于 Windows(64 位)操作系统的安装。该版本 Eclipse 支持 1.2 节所安装的 Java SE 8。

Eclipse 采用绿色安装,直接将安装包(zip 格式)解压到某个硬盘文件夹即完成了安装。例如,将已下载的安装包解压到如下文件夹:

```
C:\Java\eclipse-java-oxygen-1a-win32-x86_64
```

解压后的文件夹如图 1-18 所示,其中的 eclipse(扩展名为.exe)就是集成开发环境的主程序。

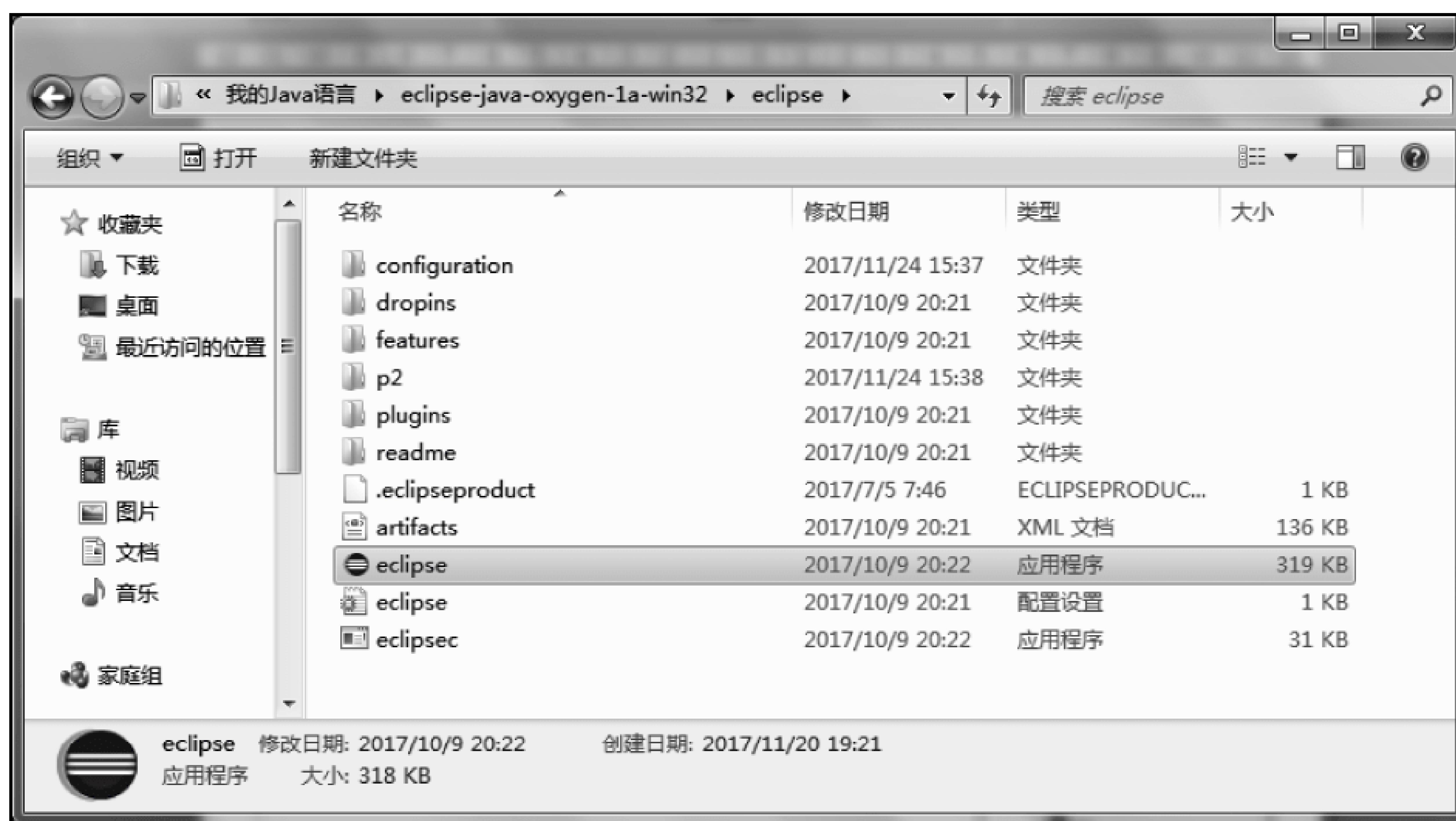


图 1-18 Eclipse 解压后的文件夹

3. 第一次运行 Eclipse

第一次运行时,Eclipse 会根据注册表自动查找 JRE 安装目录。如果 JRE 没有正确安装,则 Eclipse 会提示错误信息然后停止运行(**注:** JRE 随 JDK 一起安装,参见 1.2.3 节)。

第一次运行 Eclipse 时需要指定**工作空间**(workspace)目录(见图 1-19),该目录将作为

今后存放 Java 程序的**根目录**。在图 1-19 中单击 **Browse** 按钮，指定工作空间目录。

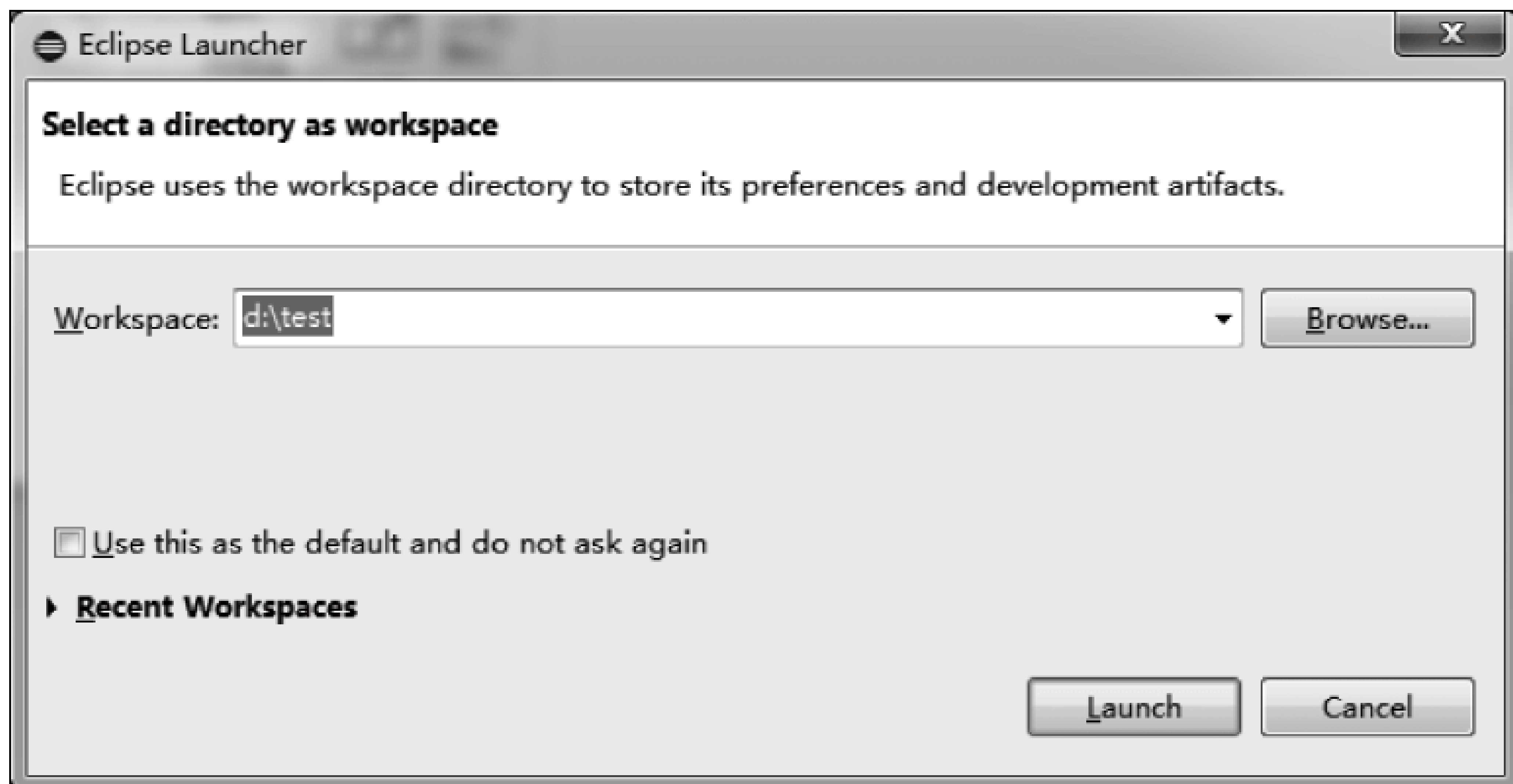


图 1-19　为"工作空间"指定目录

图 1-19 将工作空间目录指定为 d:\test，单击 **Launch** 按钮进入 Eclipse 主界面（见图 1-20）。

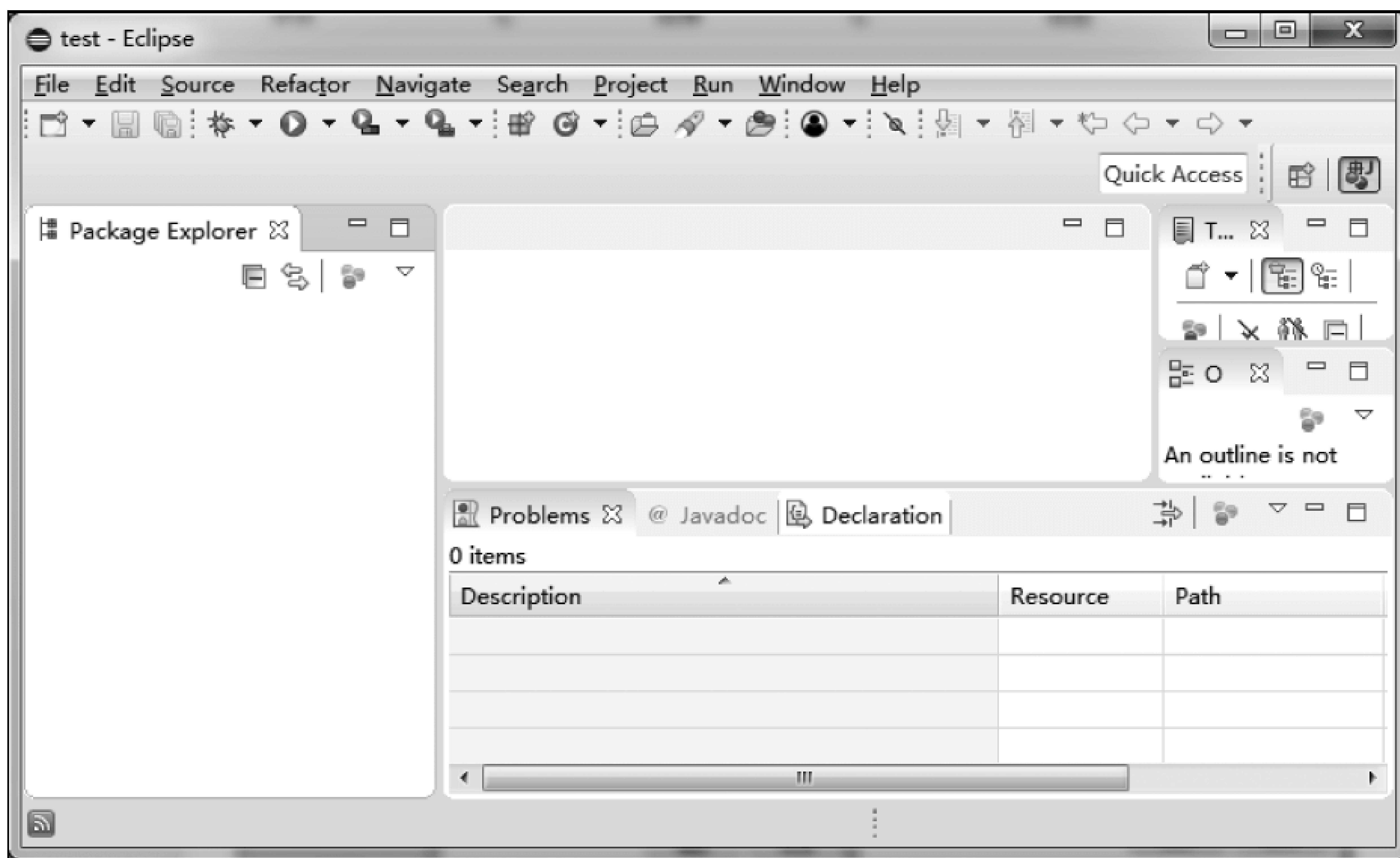


图 1-20　Eclipse 主界面

1.4.2　编写第一个 Java 程序

编写 Java 程序需要先新建 **Java 项目（project）**，然后在项目中添加 **Java 类（class）**。

- 新建一个 Java 项目，就是在工作空间目录下新建一个子目录，该子目录被称为是 Java 项目的根目录。在 Eclipse 中可以新建多个 Java 项目。
- 一个 Java 项目可以包含一个或多个 Java 类。一个类通常被保存成一个源程序文件，其文件名为类名，扩展名为.java。一个 Java 类就是一个 Java 程序。

学习 Java 语言程序设计需要进行编程练习。可以将每一章的编程练习看作是一个 Java 项目,为每一章编程练习新建一个 Java 项目。每做一个程序练习,就是在该章的 Java 项目中添加一个 Java 类,然后编写该类的程序代码。例如,为第 1 章编程练习新建一个 Java 项目 **Chapter1**,然后在该项目中编写自己的第一个 Java 程序。

1. 新建 Java 项目

在图 1-20 所示 Eclipse 主界面中选择 **File→New→Java Project**,进入新建 Java 项目对话框(图 1-21)。

为第 1 章编程练习新建一个名为 **Chapter1** 的 Java 项目,首先在图 1-21 的 **Project name**(项目名称)中输入 Chapter1,然后单击 **Finish**(完成)按钮,返回 Eclipse 主界面,这样就完成了新建 Java 项目 Chapter1 的操作。

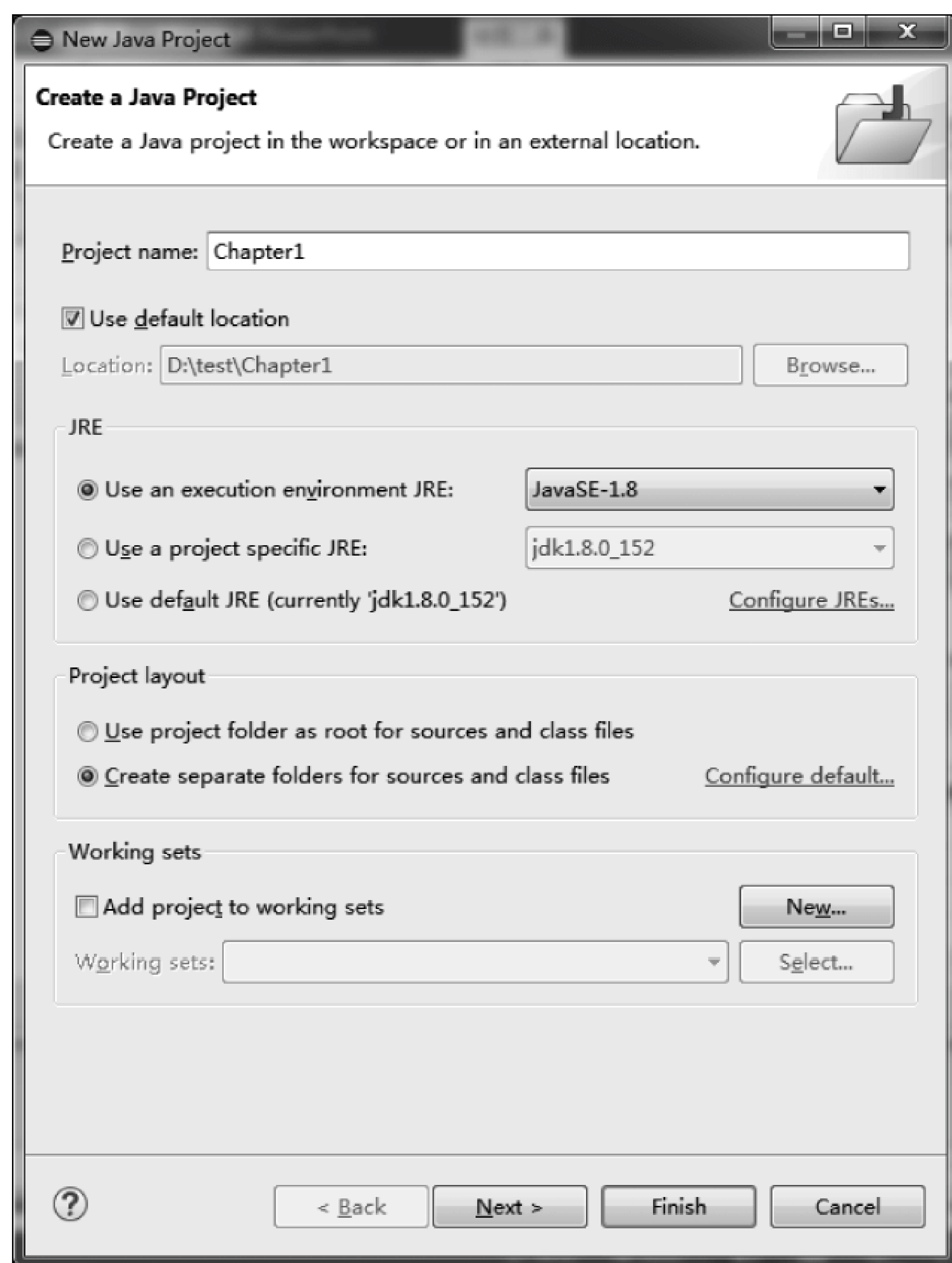


图 1-21　新建 Java 项目对话框

每新建一个 Java 项目,Eclipse 就在工作空间目录下为其新建一个子目录。例如,Eclipse 为项目 Chapter1 新建的子目录为:

```
d:\test\Chapter1
```

注:子目录名与项目名相同。

返回 Eclipse 主界面后，界面左上部的 **Package Explorer** 中将显示新建的项目 Chapter1（见图 1-22）。目前，Java 项目 Chapter1 还只是一个空项目。

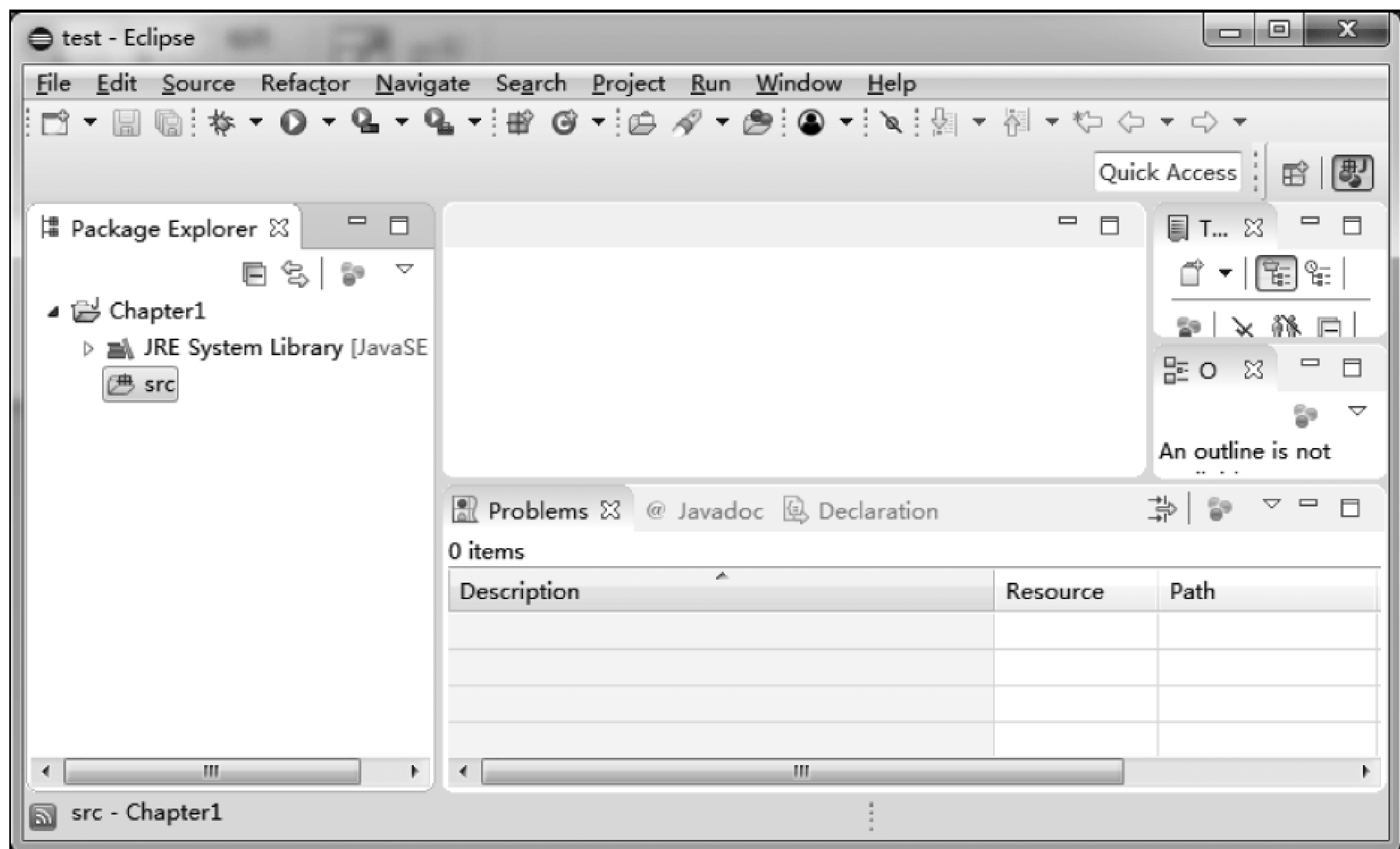


图 1-22　一个新建的 Java 项目 Chapter1

2. 在项目中新建 Java 类

在图 1-22 的界面中，先用鼠标单击 **Chapter1** 或其下面的 **src**，选中项目 Chapter1，然后在项目中添加 Java 类。一个 Java 类就是一个 Java 源程序文件，其扩展名为.java，被保存到项目目录下的 **src** 子目录中。在本例中，src 子目录的完整路径名为：

```
D:\test\Chapter1\src
```

这是一个三级目录，其中：

- **test**，这是**工作空间**目录，用于存放新建的 Java 项目(**注**：这个工作空间目录可以更改)。
- **Chapter1**，这是**项目**目录，用于存放该项目的 Java 类。该目录还包含两个子目录：src 和 bin，分别存放源程序文件(.java)和编译后的字节码程序文件(.class)。
- **src**，这是项目**源程序文件**目录，用于存放该项目下 Java 类的源程序文件(.java)。

在项目中新建 Java 类时，首先选中项目，然后选择 **File→New→Class**，进入新建 Java 类对话框(见图 1-23)。

在图 1-23 中新建 Java 类时，需关注以下几个主要设置选项。

- **Source folder**：源程序文件夹，默认为 Chapter1/src。本例使用默认文件夹。
- **Package**：Java 包名，本例保持为空。其含义将在今后讲解。
- **Name**：Java 类名，此处输入类名，例如 **p1**(或 **P1**)。
- 是否定义主函数：本例选“是”，勾选 public static void main(String[] args)复选框。

其他设置选项应保持图 1-23 中的默认状态，不要修改。单击 **Finish**(完成)按钮，返回 Eclipse 主界面，这样就完成了新建 Java 类 p1 的操作。

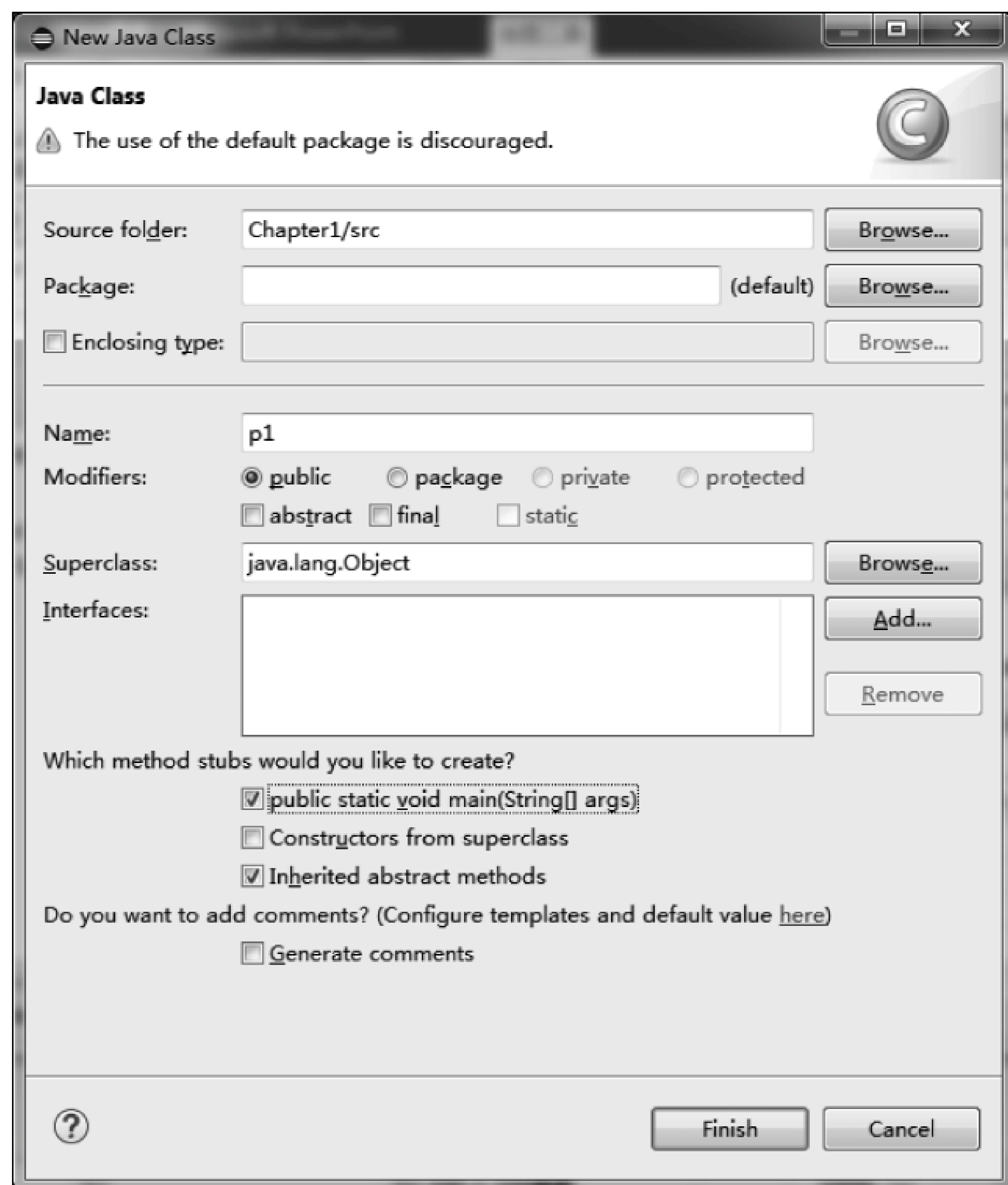


图 1-23 新建 Java 类对话框

在新建 Java 类 p1 之后，Eclipse 主界面会刷新内容，如图 1-24 所示。界面左上部的 **Package Explorer** 的项目 **Chapter1-src-(default package)** 下有一个 **p1.java**，这就是刚刚新建的 Java 类 p1 的源程序文件。

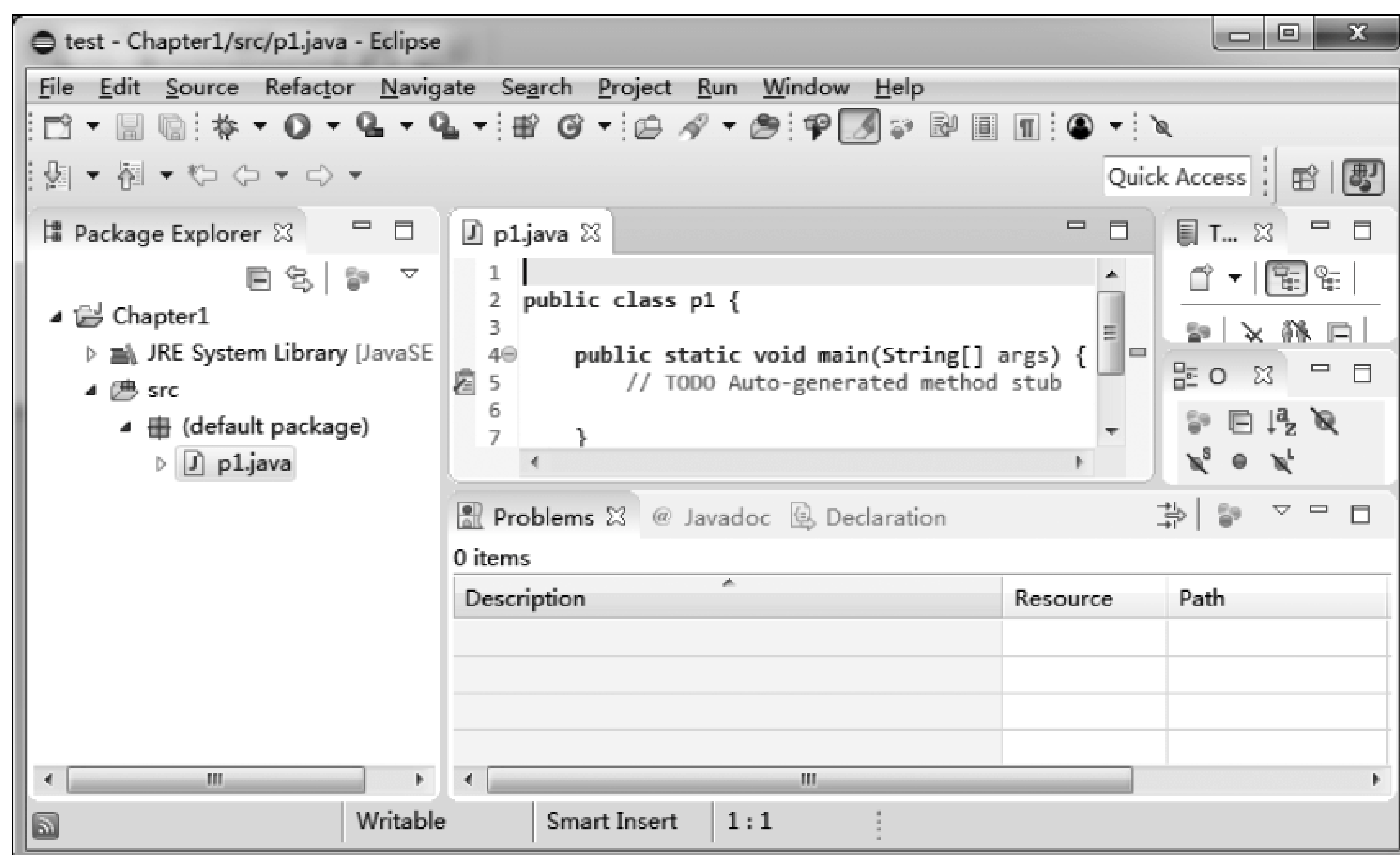


图 1-24 在项目 Chapter1 中新建了一个 Java 类 p1

新建 Java 类 p1 之后，Eclipse 在主界面的中部区域（称为“**代码编辑区**”）显示出源程序文件 p1.java 里的内容。这些内容就是类 p1 的 Java 语言代码，其中包含一个主函数 **main()**。

3. 在主函数 main()中编写代码

在主函数 main()中输入用 Java 语言编写的指令（或称为语句），例如输入一条显示问候语“Hello，World!”的输出语句：

```
System.out.println( "Hello, World!" );  //在显示器上显示：Hello, World!
```

4. 运行程序

选择菜单 **Run** 下的子菜单 **Run**，就可以运行当前打开的 Java 源程序文件中的代码（见图 1-25）。

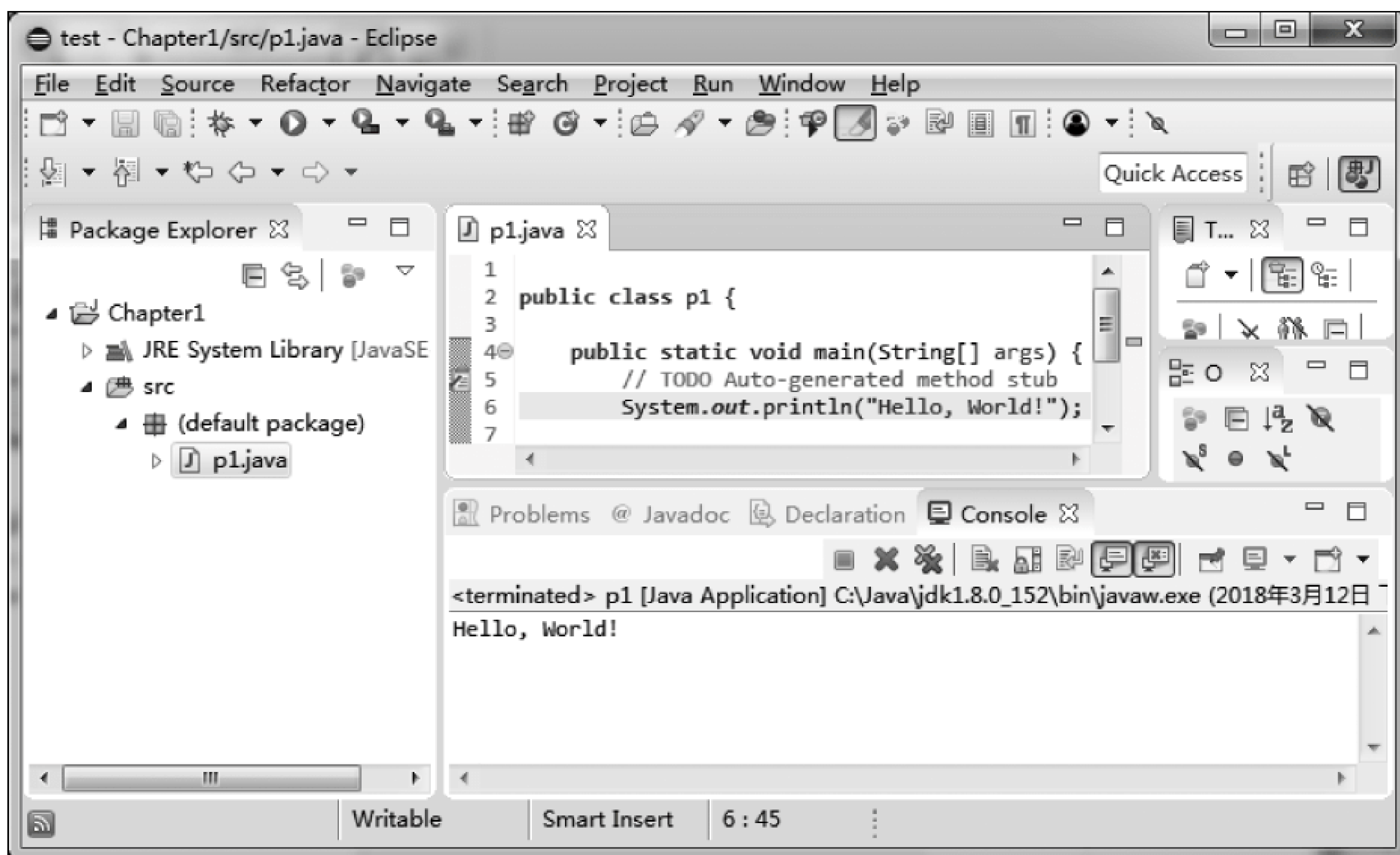


图 1-25　运行 p1.java 的程序代码

图 1-25 中，运行当前打开的 p1.java 程序代码，可以在 Eclipse 主界面的中下部区域（称为“**信息显示区**”）看到程序输出的信息内容：

```
Hello, World!
```

Eclipse 在运行程序时，会先对 Java 源程序文件进行编译，并将编译好的字节码程序保存到项目目录下的 **bin** 子目录中。例如编译源程序 p1.java，会在 bin 子目录中生成一个字节码程序文件 p1.class。运行 Java 程序，最终运行的是这个字节码程序。

Eclipse 主界面的信息显示区包含 4 个标签。初学者重点需要关注以下两个标签。

- **Console**：程序运行控制台。运行程序时，在此标签下输入原始数据，或查看程序输出结果。
- **Problems**：错误信息。在此标签下查看程序编译过程中发现的语法错误。

5. 继续编写下一个 Java 程序

可以在项目 Chapter1 中继续编写下一个程序,例如 p2.java。按照前述第 2 步的方法,再新建一个 Java 类 p2,然后编写该程序的代码(见图 1-26)。

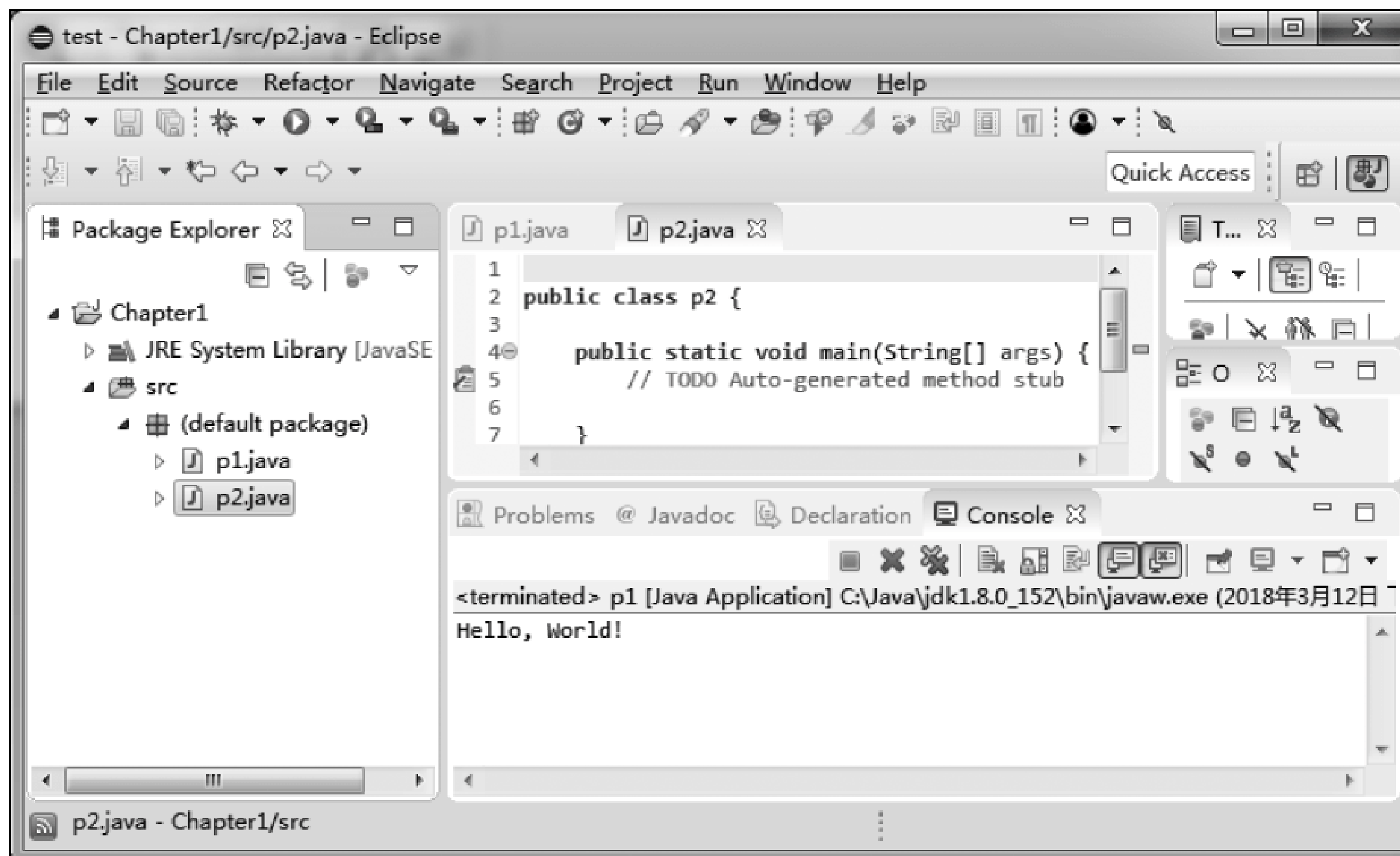


图 1-26 在项目 Chapter1 中再新建一个 Java 类 p2

本节习题

1. Eclipse 是一个非常流行的集成开发环境,它是由(　　)负责维护的。
   A. Oracle　　B. Java
   C. Eclipse Foundation　　D. Microsoft
2. 如果安装了 JDK 8,则 Eclipse 应当选择(　　)。
   A. Eclipse 3.7　　B. Eclipse 4.3　　C. Eclipse 4.5　　D. Eclipse 4.7
3. 在 Eclipse 中编写 Java 程序,第 1 步应当(　　)。
   A. 新建 Java 项目　　B. 新建 Java 类
   C. 编写 Java 代码　　D. 运行 Java 程序
4. 新建 Java 类时需设置几个主要选项,其中不包括(　　)。
   A. 源程序所在的文件夹　　B. 包名
   C. 类名　　D. 程序员姓名
5. 在 Eclipse 中运行 Java 程序,如果需要输入原始数据或查看程序输出结果,应当在 Eclipse 信息显示区的标签(　　)中进行。
   A. Problems　　B. Javadoc
   C. Declaration　　D. Console

本章学习要点

- 学习 Java 语言程序设计时，应重点学习 Java 生态圈和应用编程。
- Java 语言和 C/C++ 语言很相似，但 Java 生态圈流行开源文化，具有更多可用的类库。
- Java 语言具有自己的特点，其中最主要的特点是跨平台。
- 立即搭建 Java 语言开发环境(JDK＋Eclipse)，编写自己的第一个 Java 程序。

本章习题

1. 搭建 Java 语言开发环境。在自己的计算机上搭建并验证 Java 语言开发环境，其中包括 Java 开发包 JDK 和 Java 集成开发环境 Eclipse。

2. 模仿编程。按照 1.1.2 节中例 1-4 给出的 Java 程序代码框架，在 Eclipse 集成开发环境中编写一个计算圆面积的程序。运行程序并记录编程过程中所出现的问题及解决方法。

第2章 Java语言基础

本章讲解 Java 语言的基础语法，其中包括**数据类型**、**变量与常量**、**运算符与表达式**、**算法结构与控制语句**等。

Java 语言的基础语法与 C/C++语言比较类似，只有一些细微差别。本章会对这些差别进行**特别说明**，以便具有 C/C++语言基础的读者能快速浏览本章内容。

2.1 数据类型

为了便于硬件实现，计算机采用二进制对数据进行存储和运算。

2.1.1 计算机中的数据存储

如何在存储器中存储一个二进制数呢？计算机需要考虑两个方面的因素，它们分别是**存储位数**和**存储格式**。

1. 存储位数

计算机管理存储器的最小单位是**字节**(byte)，每个字节可存储一个 8 位二进制数。因为位数的限制，一个字节能存储的最大值为$(11111111)_2$，即十进制的$(255)_{10}$，其中下标 2 表示二进制，10 表示十进制。一个字节能存储的最小值为 0，即$(00000000)_2$。我们称，一个字节所能存储的数值范围为$(00000000)_2$～$(11111111)_2$，即十进制的 0～255。

为了管理方便，计算机以固定的位数来存储二进制数，不足部分在高位补 0。这种使用固定位数存储数据的形式被称为**定长存储**。可以将多个字节合在一起，这样能够增加存储位数，扩展可存储的数值范围。定长存储所采用的位数都是 8 的整数倍，例如 8 位(1 字节)、16 位(2 字节)或 32 位(4 字节)等，其对应的数值范围分别是 0～255、0～65535、0～4294967295。

程序所处理的数据只有存放到内存后才能被 CPU 处理，因此程序员编写程序时应首先向计算机系统申请保存数据所需的内存空间。申请内存时，需要指定存放数据所需的存储位数。存储位数越多，可存储的数值范围就越大，相应地所占用的内存空间也越大。因此程序员在编写程序时，应根据所处理数据可能的取值范围**合理地**选择存储位数。

2. 存储格式

计算机存储二进制数还需要考虑的另外一个因素是存储格式。存储格式包括以下两个

方面：以什么格式来区分**正数**和**负数**、以什么格式来区分**整数**和**实数**。

如果所处理的数据有正数，也有负数，计算机该如何存储一个数的正负号呢？假设存储位数为 8 位，可以将最高位拿出来作为**符号位**(0 表示正数，1 表示负数)，剩余的 7 位用来存放数值。这种含符号位的存储格式被称为**有符号格式**。相应地，不含符号位的存储格式被称为**无符号格式**。有符号格式可以存储正数，也可以存储负数。无符号格式则只能存储非负整数，即零或正整数。

含符号位的二进制编码形式被称为**原码**。例如，$(+82)_{10}$ 的原码是 $(01010010)_2$，而 $(-82)_{10}$ 的原码是 $(11010010)_2$。在采用有符号格式存储时，计算机使用原码的形式来存储正数，而存储负数时则使用另一种被称为**补码**的形式来存储。下面以 $(-82)_{10}$ 为例来演示负数补码的计算方法。

(1) $(-82)_{10}$ 的**原码**是 $(11010010)_2$，其中最高位为符号位，1 表示负数。

(2) 对数值部分求反，符号位不变，得到 $(10101101)_2$，这个编码被称为是 $(-82)_{10}$ 的**反码**。

(3) 将反码加 1，得到 $(-82)_{10}$ 的**补码** $(10101110)_2$。

补码与存储位数有关。存储位数不同，所转换出的补码是不同的。例如 $(-82)_{10}$ 的 16 位补码计算过程为：

$$(1000000001010010)_2 \rightarrow (1111111110101101)_2 \rightarrow (1111111110101110)_2$$

为统一起见，Java 语言做出如下规定：**正数**的补码与原码相同。这样可以说，计算机存储数据(包括正数和负数)采用的是补码格式。计算机引入补码的原因有如下两个。

(1) 定长存储时，“A－B”等于“A 的补码＋(－B)的补码”。这样可以将减法运算统一成加法运算，从而简化 CPU 的硬件设计。

(2) 采用补码存储，0 的编码是唯一的。“＋0”和“－0”的原码不同，具有二义性。如果采用 8 位存储，则它们的原码分别为 $(00000000)_2$ 和 $(10000000)_2$，而它们的补码都是 $(00000000)_2$。

无符号位与有符号位的存储格式不同，其可存储的数值范围也不同(见表 2-1)。

表 2-1　不同存储位数和存储格式情况下的数值范围

存储位数(字节数)	数值范围	
	无符号格式	有符号格式(补码)
8(1)	0～255	－128～＋127
16(2)	0～65535	－32768～＋32767
32(4)	0～4294967295	－2147483648～2147483647

计算机如何存储一个实数呢？这里先介绍一下数的**科学表示法**。例如，一组实数：

82.625，8.2625，0.82625，0.082625

它们的科学表示法分别为：

$$0.82625\times10^2,\ 0.82625\times10^1,\ 0.82625\times10^0,\ 0.82625\times10^{-1}$$

一个十进制数实数 N 的科学表示法可以写成：

$$N = M \times 10^E$$

其中，E 是指数，称为 N 的**阶码**，阶码反映了小数点的位置；M 表示 N 的全部有效数字，称

为 N 的**尾码**(或称尾数),尾码反映了数据的精度。

计算机存储实数时,先将其转换成科学表示法,然后只存储其中的阶码和尾码。这种存储实数的格式被称为**浮点格式**。下面以$(-8.2625)_{10}$为例来说明实数在计算机中的存储格式。

(1) 将$(-8.2625)_{10}$转换成浮点形式$(-0.82625\times10^1)_{10}$。

(2) 将阶码$(+1)_{10}$转换成二进制$(+1)_2$。

(3) 将尾码$(-0.82625)_{10}$转换成二进制$(-0.11010011100)_2$。**注**:只保留 11 位精度。

(4) 存储阶码和尾码的二进制编码。**注**:不同计算机的存储格式可能不同。

下面给出的演示例子用 4 位来存储阶码的补码,用 12 位来存储尾码的补码,共 16 位(占 2 字节)。

0	001	1	00101100100
阶码 符号位	阶码	尾码 符号位	尾码

3. 数据类型

计算机存储二进制数据要考虑两个因素,即存储位数和存储格式,它们共同决定了可存储的数值范围。存储非负整数可以使用无符号格式;如需存储负数则必须使用有符号格式。如需存储实数,则必须使用浮点格式,即"阶码+尾码"的存储格式。因为计算机使用定长存储,如果程序员选择不当,则保存数据时可能会出现**溢出**或**损失精度**等问题。

为了让程序员在申请内存时能方便地指定存储位数和存储格式,计算机高级语言引入了**数据类型**(data type)的概念。结合实际应用的需要,高级语言一般都预定义了若干种数据类型,并规定了每种数据类型的存储位数、有无符号位、存储整数或实数等。程序员在申请内存空间时应根据所存储数据可能的取值范围合理地选择数据类型,该数据类型决定了所申请内存空间的字节数及存储格式。Java 语言将预定义的数据类型称为**基本数据类型**(primitive data type)。

2.1.2 Java 语言中的基本数据类型

Java 语言预先定义了 8 种基本数据类型,见表 2-2。

表 2-2 Java 语言定义的 8 种基本数据类型

数据类型	说 明	存储位数(字节数)	数值范围	运 算
byte	整型	8(**1**)	$-128\sim127$,即$-2^7\sim2^7-1$	算术运算 关系运算
short		16(**2**)	$-32768\sim32767$,即$-2^{15}\sim2^{15}-1$	
int		32(**4**)	$-2^{31}\sim2^{31}-1$	
long		64(**8**)	$-2^{63}\sim2^{63}-1$	
float	浮点型(实数)	32(**4**)	$3.403\times10^{-38}\sim3.403\times10^{38}$(绝对值精度)	
double		64(**8**)	$1.798\times10^{-308}\sim1.798\times10^{308}$(绝对值精度)	
char	字符型	16(**2**)	Unicode 编码(UTF-16)	算术运算 关系运算
boolean	布尔型	未明确指定	true 或 false	逻辑运算

特别说明：Java 语言与 C/C++语言的区别(**基本数据类型**)

- Java 语言的整数类型都是有符号格式(signed)，没有无符号格式(unsigned)的整数类型。**注**：C/C++语言的整数类型既有有符号格式，也有无符号格式。
- Java 数据类型的存储位数是固定的，与操作系统或编译系统无关，其目的是为了跨平台运行。**注**：C/C++语言数据类型的存储位数与操作系统或编译系统有关。
- Java 语言的单字节整型为 byte。**注**：C/C++语言的单字节整型为 char，与字符型相同。
- Java 语言的长整型 long 占 8 字节(64 位)，是 int 型的两倍。**注**：C/C++语言中，长整型 long 与 int 型占用的字节数一样。
- Java 语言中的字符型 char 占 2 字节，保存字符的 Unicode 编码(UTF-16)。**注**：C/C++语言中的字符型 char 占 1 字节，保存字符的 ANSI 编码。
- Java 语言中布尔型的关键字是 boolean。**注**：C/C++语言中布尔型的关键字是 bool。
- Java 语言没有指针类型。例如，下列 C/C++用法在 Java 语言中是错误的。

```
int x, *p = &x;        //C/C++用法：定义一个指向变量 x 的 int 型指针变量 p
*p = 10;               //C/C++用法：通过指针变量 p 间接访问变量 x
```

本节习题

1. 每周有 7 天，为星期一到星期日分别赋予一个整数编码。使用十进制只需 1 位编码就够了，例如 0～6。使用二进制最少需要(　　)位编码。

   A. 1　　B. 2　　C. 3　　D. 4

2. 采用无符号格式，4 位二进制数可以存储的数值范围是(　　)。

   A. 0～3　　B. 1～4　　C. 0～9999　　D. 0～15

3. 计算机是以(　　)的形式来存储实数的。

   A. 原码　　B. 反码　　C. 补码　　D. 阶码＋尾码

4. 下列不同类型的数据中，存储(　　)需要使用浮点格式。

   A. 正整数　　B. 负整数　　C. 实数　　D. 文字

5. Java 语言中没有数据类型(　　)。

   A. byte　　B. unsigned int　　C. short　　D. boolean

6. Java 语言中，数据类型(　　)的存储位数与 char 类型一样多。

   A. byte　　B. short　　C. int　　D. double

7. Java 语言中，数据类型(　　)的存储位数与 boolean 类型一样多。

   A. byte　　B. short　　C. int　　D. 不确定

8. Java 语言中，数据类型(　　)的存储位数与 long 类型一样多。

   A. byte　　B. short　　C. int　　D. double

2.2 变量与常量

第1章曾介绍过一个温度换算程序。摄氏温度到华氏温度的换算公式是：

$$f = c \times 1.8 + 32$$

在这个换算公式中，f 和 c 是变量，而1.8和32是常量。计算机语言也是通过变量或常量来表示数据的，但其含义与我们的常识存在一些区别。

2.2.1 变量

数据是程序处理的对象。数据要存放在内存中才能被CPU读取和处理，处理后的结果也只能保存回内存中。程序中的数据包括原始数据、中间结果、最终结果等，Java语言使用**变量**(variable)来保存这些数据。

定义变量就是为变量申请内存空间。定义变量后，可以向该变量所分配的内存单元**写入**(write)数据或**读出**(read)其中的数据，这被称为是**访问变量**。

程序执行时，程序中的变量就对应内存中的某个内存单元。程序结束退出时，变量将释放其所占用的内存单元，以便给其他程序继续使用。简单地说，程序中的**变量**＝**内存单元**。

程序员在定义变量时要考虑3方面的内容。

- 变量如何在内存中存储(存储位数和存储格式)？程序员需要指定变量的数据类型。
- 变量如何命名？程序员需要了解Java语言的命名规则。
- 如何编写定义变量语句？程序员需要了解定义变量语句的语法。

1. 为变量选择数据类型

程序员应根据所处理数据**可能的**取值范围来**判断**应定义哪种类型的变量。所依据的原则是：既要保证精度、防止溢出，又要尽可能少地占用内存。

例如，月份可以用整数表示，其数值范围为1～12。定义一个保存月份数据的变量，程序员应当选择哪种数据类型呢？选择byte、short、int或long这4种整数类型都可以。但选择byte类型最合理，因为byte类型既能满足月份数据数值范围的需要，同时所占用的字节数又最少(仅占1字节)。

在温度换算公式 $f=c\times1.8+32$ 中，f 是华氏温度，c 是摄氏温度。温度数据通常是实数，因此定义保存温度数据的变量应选择实数类型，即float或double类型。通常，float类型能够满足温度数据的精度要求，但其所占用内存的字节数为4字节，是double类型的一半，因此选用float类型更加合理。

2. 为变量命名

Java语言的词法元素包括关键字、标识符、常量、运算符、分隔符等。**关键字**(keyword)是Java语言预先保留的具有特定含义的单词。例如，表2-2中表示基本数据类型所用到的单词int、float、double等，它们就是Java语言的关键字。下面列出了Java语言所保留的51个关键字。

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	null	package	private	protected
public	return	strictfp	short	static
super	switch	synchronized	this	throw
throws	transient	try	void	volatile
while				

程序中所包含的一些实体(例如变量)需要程序员为它们命名。由程序员定义的程序实体名称被统称为**标识符**(identifier)。在Java语言中,对标识符的命名需符合如下命名规则。

- 以大写或小写英文字母、下画线"_"、美元符号"$"开头。
- 由大写或小写英文字母、下画线"_"、美元符号"$"、数字0~9组成。
- 不能是关键字。

例如,abc、Abc、_bc、abc123、abc_123、A、a、$No1等,符合标识符命名规则。123、abc.123、温度、float等,不符合标识符命名规则,属于语法错误。

另外,Java语言区分大小写英文字母。例如,abc和Abc是两个不同的标识符。

3. 定义变量语句的语法

Java语法:**变量定义语句**

```
数据类型 变量名 1, 变量名 2, …, 变量名 n ;
```

语法说明:

- **数据类型**指定了变量的存储位数和存储格式。
- **变量名**需符合标识符的命名规则。
- 可在一条语句中定义多个具有相同数据类型的变量,变量之间用","隔开。

举例:定义两个变量ctemp和ftemp

```
double ctemp ;          //计算机为 double 型变量 ctemp 分配 8 个连续的字节作为其内存单元
                        //并将以浮点格式在该内存单元中存储数据
double ftemp ;
```

或

```
double ctemp, ftemp ; //在一条语句中定义两个 double 型的变量
```

程序由程序员编写,由计算机执行。当执行到程序中的定义变量语句时,计算机将根据数据类型为变量分配指定字节个数的内存单元。后续程序将通过变量名来访问这个内存单

元,例如向其中写入数据,或读出其中的数据。

新定义变量的内存单元中还未曾写入过数据,此时其数值被标记未 **null**,其含义是数值为空,即无效数据。null 是 Java 语言的关键字。

高级语言通过变量名来访问内存单元,变量名便于程序员记忆和使用。高级语言程序需编译成机器语言程序才能被计算机硬件识别和执行。机器语言是通过地址访问内存的。编译时,程序中的变量名被转换成了内存地址。换句话说,程序员在编写高级语言程序时使用变量名来申请和访问内存,而计算机执行其编译后的机器语言程序时使用的则是内存地址。

4. 访问变量的内存单元

定义变量后,可以向变量所分配的内存单元写入数据或读出其中的数据。

1) 写入数据

Java 语言向变量内存单元写入数据的操作有 3 种方式。

- 从键盘**输入**数据。例如:

```
Scanner sc = new Scanner( System.in );      //创建键盘扫描器
ctemp = sc.nextDouble();                    //从键盘接收数据并写入变量 ctemp 的内存单元
```

- 使用**赋值**运算符"="(即等号),对变量进行赋值运算。例如:

```
ctemp = 30 + 6;
```

该语句先计算等号右边的表达式,然后将计算结果(36)赋值给等号左边的变量 ctemp,即将数值 36 写入变量 ctemp 的内存单元。

- 定义时**初始化**。定义变量的同时为变量赋初始值,这就是初始化。例如:

```
int x = 10, y;
```

该语句定义了两个 int 型变量 x 和 y。执行该语句时,计算机为变量分配内存空间。x 被初始化了,计算机在为 x 分配内存单元的同时向该内存单元写入初始值 10。y 没有被初始化,其内存单元中的数值为 null。

2) 读出数据

Java 语言从变量读出数据的操作有两种方式。

- 当变量作为**操作数**参与运算时,计算机将自动读取其内存单元中存放的数据。例如:

```
ftemp = ctemp * 1.8 + 32;
```

该语句中等号右边的变量 ctemp 是作为操作数参与运算的,计算机会自动读取其内存单元中的数据。读出数据后,再用该数据进行运算,并将运算所得到的结果赋值给等号左边的变量 ftemp。

- 使用**输出**语句读出并显示变量内存单元中存放的数据,以便用户查看。例如:

```
System.out.println( ftemp );
```

该语句指示计算机读出变量 ftemp 内存单元中存放的数据,并在显示器上显示出来。

定义后的变量才有内存单元，才能被访问。程序员在编写 Java 程序时应遵循**“先定义，后访问”**的原则。未经定义的变量不能访问。另外，不能读取数值为 **null** 的变量，否则属于语法错误。

2.2.2　常量

这里仍以温度换算公式 $f=c\times1.8+32$ 为例，具体讲解什么是程序设计中的变量和常量。

变量是在编写程序时不能确定其数值大小的量，例如摄氏温度 c 是今后程序执行时由用户输入的。程序员需要在编写程序时，使用定义变量语句预先为变量分配好内存单元。例如为摄氏温度定义一个变量 ctemp，这样程序执行时才能有内存单元，并能在其中存放摄氏温度数据。

而**常量**(constant)则是在编写程序时就能确定其数值大小的量，例如 1.8 和 32。程序员可以将数值直接书写在程序代码中，它们也被称为**字面常量**(literal constant)。不同数据类型的常量有不同的书写形式。

1. 整数常量

十进制：20、－20。

八进制：020、－020。

十六进制：0x20、－0X20。

二进制：0b10100、－0B10100。**注**：C/C++语言没有这种二进制书写形式。

整数常量默认为 int 型。可以添加后缀 L(大小写都可以)将其转为 long 型。例如：

20L、－20l

推荐使用大写字母 L，因为小写字母 l 容易与数字 1 混淆。

2. 实数常量(浮点常量)

带小数点：20.5、－20.0。**注**：－20 是整数，而－20.0 则是实数。

科学记数法：2.05E1 或 0.205E2、－2.0E1 或－0.2E2。**注**：E 大小写都可以。

实数常量默认为 double 型，可以添加后缀 F(大小写都可以)将其转为 float 型。例如，20.5f、2.05e1F。

3. 字符常量

可见字符：'A'、'a'、'1'、'中'。

转义字符：'\uxxxx'。**注**：u 表示 Unicode 编码，xxxx 是字符的码值(十六进制)。

Java 语言预定义的转义字符：'\n'、'\t'、'\b'、'\f'、'\r'、'\"'、'\''、'\\'等。

Java 语言保存一个字符需要占用 2 字节，所保存的是该字符的 Unicode 编码(UTF-16)。和英文字符一样，一个汉字字符也算是一个字符。

注：在 C/C++语言中，一个英文字符占 1 字节；一个汉字字符占 2 字节，算是两个字符。英文字符保存的是其 ASCII 编码，汉字字符保存的是其 GBK 编码。这种中英文混合编码方式被称为 ANSI 编码。

4. 字符串常量

可见字符的字符串："Abc"、"中国 China"、"A"、""(空字符串)。

带转义字符的字符串如下。

"中国\nChina"　　注：字符串中包含一个换行。

"\"中国\""、"\'China\'"　　注：字符串中包含双引号和单引号。

"C:\\Example\\test.java"　　注：字符串中包含反斜杠

5. 布尔常量

布尔常量只有两个：**true**(真)、**false**(假)。

2.2.3 只读变量

如果程序所处理的某个数据是常量，在程序运行过程中不需要变动，则可以定义一个只读(read-only)**变量**来保存该数据。

Java 语法：**定义只读变量**

```
final 数据类型 只读变量名 = 初始值;
```

语法说明：

- 使用关键字 **final** 定义只读变量。
- 只读变量只能被赋值一次。只读变量在取得初始值之后，只能进行**读取**操作，不能做**写入操作**(例如再次赋值)。
- 定义只读变量时通常都会**初始化**。

举例：

```
final int x = 5;      //定义只读变量 x,初始值设定为 5
x = 10;               //语法错误:不能对只读变量 x 再次赋值
final int y;          //定义只读变量 y 时没有初始化,此时其数值为 null
y = 5;                //正确:第一次为只读变量 y 赋值
y = 5;                //语法错误:不能对只读变量 y 再次赋值,即使是赋同样的值
```

只读变量从本质上讲是一个变量，从功能上看就是用变量实现了常量的功能。只读变量有时也被称作常变量，或简单称作常量。和字面常量相比，只读变量具有可以提高程序可读性、便于调整常量值等优点。

特别说明：Java 语言与 C/C++语言的区别(**变量与常量**)如下。

- Java 变量名可包含美元符号 $ 。**注**：C/C++语言不可以。
- 未初始化的 Java 变量是 null，不能读取。**注**：C/C++语言可以，但读取的是随机值。
- Java 语言可以书写二进制整数常量。**注**：C/C++语言不可以。
- Java 语言以 Unicode 编码(UTF-16)存储字符，一个汉字也是一个字符。**注**：C/C++语言以 ANSI 编码存储字符，一个汉字相当于是两个字符。
- Java 语言没有“符号常量”，但可通过“只读变量”实现对应的功能。**注**：C/C++语言可以使用“#define”宏定义指令定义符号常量。

本节习题

1. 假设变量 x 的值域为[0,50000]的整数，则其最适合的数据类型是(　　)。
   A. short　　B. int　　C. long　　D. float
2. 假设变量 x 的值域为[−1.0,1.0]的实数，则其最适合的数据类型是(　　)。
   A. short　　B. int　　C. long　　D. float
3. 下列名称中，(　　)可以作为变量名。
   A. No.1　　B. 123_ABC　　C. long　　D. Long
4. 下列定义变量语句中，错误的是(　　)。
   A. int x,y;　　B. int x=5,y;　　C. int x=5,y=5;　　D. int x=y=5;
5. Java 源程序中，常量(　　)的数据类型是 float 型。
   A. 10　　B. 10L　　C. 10.0　　D. 10.0f
6. Java 源程序中，整数(　　)的数值最小。
   A. 15　　B. 15L　　C. 015　　D. 0x15
7. 计算圆形周长的公式是：周长=$2\pi r$，其中 r 为半径。编写计算圆形周长的程序时需要将(　　)定义成变量。
   A. π　　B. 半径　　C. 周长　　D. 半径和周长
8. 计算圆形周长的公式是：周长=$2\pi r$，其中 r 为半径。编写计算圆形周长的程序时可以将(　　)定义成常量。
   A. π　　B. 半径　　C. 周长　　D. 2 和 π

2.3 运算符与表达式

描述计算内容和计算过程的公式被称为**表达式**(expression)。表达式由**运算符**(operator)、**操作数**(operand)和**括号**(parentheses)组成。Java 语言中，在表达式后加分号“;”就构成了一条完整的**表达式语句**。表达式语句用于处理数据。

运算符有**优先级**，优先级高的先算。同级运算符按其**结合性**(从左到右或从右到左)所规定的顺序来计算。**括号**可以提高优先级，括号内的先算，多层括号时先算里层括号。某些运算符需要两个操作数(例如加法运算)，称为**双目**运算符；某些运算符只需要一个操作数，称为**单目**运算符。

Java 语言根据功能和用途将运算划分为**算术运算**、**位运算**、**关系运算**、**逻辑运算**等不同类型。本节先介绍算术运算和位运算，后面再介绍关系运算和逻辑运算。

2.3.1 算术运算

加、减、乘、除是最常用的算术运算，Java 语言分别用不同的符号来表示它们：+(加)、−(减)、*(乘)、/(除)，这些符号被称为**算术运算符**。由算术运算符构成的表达式称为**算术表达式**。Java 语言中加、减、乘、除运算的含义与我们的常识是一致的，但也存在一些区别。

1. 优先级和结合性

运算符有不同的优先级，优先级高的先算。同级运算符按其结合性(从左到右或从右到左)所规定的顺序来计算。Java 语言中，不同运算符有不同的结合性。例如，+、-、*、/等大多数运算符的结合性是从左到右，也有某些运算符的结合性是从右到左。表 2-3 列出 Java 语言中各运算符的优先级和结合性。

表 2-3 Java 运算符的优先级和结合性

高优先级	[] . ()(注：下标、成员、调用)	从左到右
↓	++ -- + - ~ !(注：自增、自减、正负号、位反、非)	从右到左
	* / %	从左到右
	+ -	从左到右
	<< >> >>>	从左到右
	< > <= >= instanceof	从左到右
	== !=	从左到右
	&	从左到右
	^	从左到右
	|	从左到右
	&&	从左到右
	||	从左到右
	? :	从右到左
低优先级	= += -= *= /= %= &= ^= |= <<= >>= >>>=	从右到左

2. 操作数及其数据类型转换

算术表达式中，参与运算的操作数可以是常量、变量等。Java 语言中，一个操作数除了代表一个数值，还具有特定的数据类型。例如，表达式"5 + 3"是含两个操作数(常量)的加法运算。这两个操作数的数值分别是 5 和 3，数据类型都是 int 型。

在 Java 语言中，当不同类型的两个操作数参与算术运算时，要先转换成相同类型，然后再进行计算。Java 语言为数据类型转换提供了**强制转换**和**自动转换**两种方法。

1) 强制转换

数据类型强制转换就是由程序员主动将操作数由一种数据类型转换成另一种数据类型。

Java 语法：**数据类型强制转换**

```
(数据类型) 操作数
```

或

```
(数据类型)(操作数)
```

举例：

```
(short)32 指定 32 为短整型(2 字节)          (long)(-32)指定 -32 为长整型(8 字节)
(float)1.8 指定 1.8 为单精度浮点型(4 字节)   (double)1.8 指定 1.8 为双精度浮点型(8 字节)
```

注：数据类型应与操作数的数值相符，否则将造成数值的改变。例如：

```
(float)32 将 32 变为 32.0(可以接受)          (int)1.8 将 1.8 变为 1(丢失小数部分)
(byte)129 将 129 变为 - 127(溢出)            (short)32769 将 32769 变为 - 32767(溢出)
```

注：为什么(byte)129 会变成－127 呢？因为 129 的二进制是$(10000001)_2$，使用单字节 byte 存储时，其最高位被当作符号位(1 表示负数)，并按补码的格式来解读。$(10000001)_2$ 正是－127 的补码。类型 byte 的数值范围是－128～＋127，因此 129 超出了该存储范围，这就是溢出。

程序员在 Java 程序中编写算术表达式时，应合理运用数据类型强制转换。例如表达式"5.5＋3"，其中 5.5 是 double 型(Java 语言默认带小数点的数都是 double 型)，3 是 int 型。可以将 3 转换为(double)3，使两个操作数都为 double 型，即"5.5＋3.0"。也可以将 5.5 转换成(int)5.5，使两个操作数都为 int 型。但第二种转换会丢失 5.5 的小数部分，即"(int)5.5＋3"转换后等价于"5＋3"，通常这是不可接受的。

2）自动转换

程序员在 Java 程序中编写算术表达式时，可以将数据类型转换的工作交由编译器完成。Java 编译器在编译源程序时，如果发现某个算术表达式含有不同类型的操作数，则进行自动转换。自动转换(或称为隐含转换)的原则是"**将低类型向高类型转换**"。Java 语言中各数值类型(整数和实数)的高低顺序如下。



数据类型越高，其可存储的数值范围越大(因为占用字节数多)，精度也越高，因此这种自动转换是安全的。例如表达式"5.5＋3"，编译器编译时会自动将 3(int 型，低类型)转换为 double 型(高类型)，使两个操作数的类型一致，即都为 double 型。"5.5＋3"经过自动转换，它等价于"5.5＋3.0"。

Java 语言的数据类型自动转换功能可以减轻程序员的工作量。

3. 表达式结果

在 Java 语言中，任何数据都是有数据类型的，因此表达式的计算结果有值，也有数据类型。算术表达式计算结果的数据类型等于其操作数的数据类型。例如，算术表达式

$$5+3$$

该表达式计算结果的数值等于 8，数据类型应当为 int 型。因为操作数 5 和 3 的数据类型都是 int 型，所以表达式结果的数据类型也为 int 型。

如果参与运算的操作数类型不同，则进行自动转换。例如，算术表达式

$$5.5+3$$

该表达式计算结果的数值等于 8.5，数据类型为 double 型。其中操作数 5.5 是 double 型，3 是 int 型。遵循"低类型向高类型转换"的原则，3 被自动转换为 double 型。转换后，两个操作数的类型都是 double 型，因此表达式结果的数据类型也为 double 型。

两个整数类型相除将丢失小数。例如，算术表达式

$$5/2$$

该表达式计算结果的数值不是2.5,而是2。因为参与运算的两个操作数都是int型,所以表达式结果的数据类型也是int型,其数值将丢掉小数部分而只保留整数部分。将操作数改为double或float类型可以避免上述丢失小数的问题。例如,程序员可以将表达式修改成如下形式:5.0/2、5/2.0、(double)5/2、5/(float)2。

4. 括号

在表达式中,括号可以提高优先级。括号内的先算,多层括号时先算里层括号。例如表达式:

(3*(2+5)-1)/2

与数学上不同的是,Java表达式只使用小括号"()",有多层括号时也是这样。Java语言对中括号"[]"和大括号"{}"分别赋予了新的含义,被用在了其他场合。

2.3.2 其他算术运算符

Java语言还有几个比较特殊的算术运算符。

1. 取正/取负运算符+和-

Java语言中的取正/取负运算符就是数学上所说的正负号,它对其后的操作数取正或取负。取正/取负运算符是单目运算符,即只有一个操作数。可以将+32、-32、-x等理解成是一个由取正/取负运算符构成的算术表达式。例如"-x"是一个算术表达式,该表达式结果的数据类型与变量x类型相同,数值等于变量x中所保存数值的负值。

2. 取余运算符%

取余运算是计算两个操作数相除后得到的余数。例如,10÷6的整数商等于1,余数等于4,因此10 % 6的结果为4。取余运算符%只能对两个整型操作数进行取余运算,运算结果也是整型。%属于算术运算符,其优先级和结合性与乘除运算符相同。

3. 自增运算符++

如果想把某个数值型变量x的值加1,可使用自增运算符++。例如x++,计算机计算该表达式时,先读出变量x的值,将其加1后再重新写回x的内存单元。++是单目运算符,操作数必须是变量。

x++还是一个由自增运算符++构成的表达式。该表达式的结果等于x加1之前的值,数据类型与x的类型相同。

++是一种**泛化的**运算符。与普通运算符相同的是,泛化运算符与操作数一起构成表达式,表达式的结果可以作为操作数继续参与下一步运算。与普通运算符不同的是,泛化运算符在运算的同时还会修改参与运算操作数的值。例如,加、减、乘、除从来不会修改操作数的值,而x++在计算表达式结果的同时还会修改操作数x的值。Java语言中类似的运算符还有下面将要介绍的自减运算符--、赋值运算符=等。

将++放在变量之后,这被称为是**后置**形式的自增运算符。也可以将++放在变量之前,这被称为是**前置**形式的自增运算符。将自增运算符前置或后置,其所构成表达式的结果

不同：后置表达式和前置表达式都能将变量的值加 1，但后置表达式的结果等于该变量加 1 之前的值，而前置表达式的结果等于变量加 1 之后的值。

例如，已有变量 x 且“int x=10;”，则 x++ 和 ++x 都能将 x 的值加 1，变成 11。但表达式 x++ 的结果为 10，而表达式 ++x 的结果为 11。表达式结果对该表达式继续参与下一步计算是有意义的，例如表达式(x++) * 2 的结果等于 20，而表达式(++x) * 2 的结果等于 22。

4. 自减运算符 --

自减运算符与自增运算符类似，只是将加 1 操作变成减 1 操作。自减运算符也有后置与前置两种形式，例如 x--(后置)或 --x(前置)。

2.3.3 位运算

计算机程序可以用一个**二进制位**来记录某种对象的开关状态，这种二进制位被称为**状态位**。举个例子，假设用计算机来控制一组电灯(用 1 表示开，0 表示关)，则一个字节可以表示 8 盏电灯的开光状态，两个字节就可以表示 16 盏电灯的开关状态。对状态位设定就可以控制某盏电灯的开关。Java 语言提供了 7 种位运算符，可应用于状态位的设定或检测，它们被统称为**位运算符**。

1. 位反运算符~

位反运算符是单目运算符，其运算规则是：**将 1 变成 0，0 变成 1**。根据数据类型的不同，程序中参与位反运算的操作数至少有 8 位(例如 byte 型)。位反运算是将操作数中的所有位同时进行取反。例如一个 8 位的位反运算：

```
~   0101 0101
=   1010 1010
```

在实际应用中，位反运算可将操作数中的所有状态位同时进行反置。假设定义一个 byte 型变量 s 来记录 8 盏电灯的开关状态：

```
byte s = 0x55;    //0x55 是十六进制数，其对应的二进制为(01010101)₂
```

对变量 s 进行位反运算可将 8 盏电灯中原来亮着的灯关闭，原来没亮的灯打开。用 Java 语言来描述上述位反运算，其形式为：

```
s = ~s;
```

2. 位与运算符&

位与运算符是双目运算符，其运算规则是：**参与运算的两个位都为 1，则结果为 1，否则为 0**。参与位与运算的两个操作数是按位进行运算。例如一个 8 位的位与运算：

```
        00110011
&       00001111
=       00000011
```

位与运算可应用于检测操作数中某个状态位的状态，或将其置为0，此时另一个操作数被称为**掩码**(mask)。假设一个byte型变量s，如想检测s中某一位的状态是0还是1，则可使用与该位对应的掩码进行位与运算。例如：

```
        bbbbbbbb        操作数s,其中b表示0或1
&       00000010        检测倒数第2位状态的掩码(0x02)
=       000000b0        运算结果:保留倒数第2位,其他位变成0
                        如果位与结果为0(即8位全部为0),则倒数第2位的状态为0;
                        否则倒数第2位的状态为1
```

用Java语言来描述上述位与运算，它是一个位与运算表达式，其形式为：

```
s & 0x02
```

位与运算还可以将变量s中某一位的状态置0。例如：

```
        bbbbbbbb        操作数s,其中b表示0或1
&       11111101        将倒数第2位状态置0的掩码(0xFD)
=       bbbbbb0b        运算结果:将倒数第2位置成0,其他位不变
```

用Java语言来描述将变量s倒数第2位置0的操作，它是一条含位与运算的表达式语句，其形式为：

```
s = s & 0xFD;
```

3. 位或运算符|

位或运算符是双目运算符，其运算规则是：**参与运算的两个位只要有一位为1，则结果为1，否则为0**。参与位或运算的两个操作数也是按位进行运算。例如一个8位的位或运算：

```
        00110011
|       00001111
=       00111111
```

位或运算可用于将操作数中的某个状态位置为1。例如，假设一个byte型变量s，则可选择掩码0x02将其倒数第2位的状态置为1。

```
        bbbbbbbb        操作数s,其中b表示0或1
|       00000010        将倒数第2位状态置1的掩码(0x02)
=       bbbbbb1b        运算结果:将倒数第2位置成1,其他位不变
```

用Java语言来描述上述位或运算，它是一条含位或运算的表达式语句，其形式为：

```
s = s | 0x02;
```

4. 异或运算符^

异或运算符是双目运算符，其运算规则是：**参与运算的两个位不同(0和1，或1和0)，则结果为1，否则为0**。参与异或运算的两个操作数按位进行运算。例如一个8位的异或运算：

```
        00110011
^       00001111
=       00111100
```

异或运算可用于将操作数中的某个状态位进行反置，即原来为 0 则反置成 1，原来为 1 则反置成 0。例如，假设一个 byte 型变量 s，则可选择掩码 0x02 将其倒数第 2 位的状态进行反置。

```
        bbbbbb0b      操作数 s，其中 b 表示 0 或 1. 假设倒数第 2 位为 0
^       00000010      将倒数第 2 位状态进行反置的掩码(0x02)
=       bbbbbb1b      运算结果：将倒数第 2 位由 0 反置成 1，其他位不变

        bbbbbb1b      操作数 s，其中 b 表示 0 或 1. 假设倒数第 2 位为 1
^       00000010      将倒数第 2 位状态进行反置的掩码(0x02)
=       bbbbbb0b      运算结果：将倒数第 2 位由 1 反置成 0，其他位不变
```

用 Java 语言来描述上述异或运算，它是一条含异或运算的表达式语句，其形式为：

```
s = s ^ 0x02;
```

5. 左移运算符<<

左移运算将操作数按二进制位左移指定的位数，左移时高位被移除，低位补 0。例如将一个 8 位操作数左移 2 位：

```
        00110011      8 位操作数
<<      2             左移 2 位
=       0011001100    高 2 位被移除，低 2 位补 0，得到 11001100
```

左移运算符的语法形式是：

```
操作数<<左移位数
```

假设一个整型变量 s，用 Java 语言来描述将 s 左移 2 位的语法形式是：

```
s << 2
```

6. 右移运算符>>和>>>

右移运算将操作数按二进制位右移指定的位数，右移时低位被移除。高位怎么补呢？>>是**带符号**右移，高位补符号位；>>>是**不带符号**右移，高位补 0。例如将一个 8 位操作数右移 2 位（带符号右移）：

```
        10110011      8 位操作数，最高位为符号位(1 表示负数)
>>      2             带符号右移 2 位
=       1110110011    低 2 位被移除，高 2 位补符号位(1)，得到 11101100
```

如果使用不带符号右移，则右移 2 位的结果如下：

```
        10110011      8 位操作数，最高位为符号位(1 表示负数)
>>>     2             不带符号右移 2 位
=       0010110011    低 2 位被移除，高 2 位补 0，得到 00101100
```

右移运算符的语法形式是：

操作数>>右移位数 注：带符号右移。

操作数>>>右移位数 注：不带符号右移。

假设一个整型变量 s，用 Java 语言来描述将 s 右移 2 位的语法形式是：

```
s >> 2
```

或

```
s >>> 2
```

需要注意的是，所有参与位运算的操作数只能是整型(byte、short、int 或 long)。如果对其他类型(例如 double)的操作数进行位运算，编译时会提示语法错误。

2.3.4 赋值运算

1. 赋值运算符 =

赋值(assignment)运算符＝用于修改变量的数值，即将新数值写入变量对应的内存单元，存储在该内存单元中的原数值将被覆盖。例如，对下面例子中的变量 x 和 y 赋值：

```
int x = 0, y = 0;
x = 5;
y = x + 3;
```

赋值运算符的作用是将＝右边表达式的结果赋值给左边的变量。例子中的变量 x、y 的初始值都为 0。赋值后，x 的值变成 5，y 的值变成 8。

语法规则上，赋值运算符＝的左边必须是变量，右边的可以是常量、变量或表达式。常量或变量可以理解成一个最简单的表达式。

赋值运算本身也构成一个**赋值表达式**。该表达式结果的数据类型与左边变量的类型相同，数值等于左边变量赋值以后的数值。上例中，“x = 5”构成一个赋值表达式，其结果的类型为 int 型(即 x 的数据类型)，数值为 5(即 x 赋值以后的数值)。赋值表达式可以继续参与运算。例如，“(x = 5) * 2”的结果等于 10。

赋值运算符的优先级最低，结合性为从右到左。例如，混合运算“y＝x＝2＋6”与“y＝(x＝(2＋6))”等价。因为加法优先级高，先算“2＋6”得到 8；两个赋值运算符按从右到左的次序先算“x＝8”(结果为 8)；最后再算“y＝8”(结果也为 8)。计算机执行语句“y＝x＝2＋6；”后，变量 x 和 y 都被赋值为 8。

Java 语言中，加、减、乘、除这样的普通运算符在运算时不会改变操作数的值。而赋值运算符＝和自增运算符＋＋、自减运算符－－等则属于泛化的运算符，由它们构成的表达式在产生运算结果的同时还会改变操作数的值。合理运用泛化运算符可以让语句更加简洁。例如：

语句：`a = 10;  b = 10;  c = 10;` 可简写成：`a = b = c = 10;`

语句：`y = x;  x = x + 1;` 可简写成：`y = x++;`

语句：`x = x + 1;  y = x;` 可简写成：`y = ++x;`

```
语句：y = x;  x = x - 1;          可简写成：y = x--;
语句：x = x - 1;  y = x;          可简写成：y = --x;
```

2. 复合赋值运算符

赋值运算符＝还可以与部分算术运算符和位运算符组成**复合赋值运算符**共 11 个。复合赋值运算符有+=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=、>>>=。

"x ?= **exp**"等价于"x = x ? (**exp**)"，其中，? 表示某个运算符，x 是一个变量，exp 是一个表达式。"x ?=exp"实际上是一种简写形式。例如：

```
x = x + 5;
```

可简写成：

```
x += 5;
```

计算机执行该语句的过程是：先读出变量 x 的值，与 5 进行加法运算，然后再将运算结果写回 x 对应的内存单元。另外，复合赋值运算符总是先计算右边的表达式。例如：

```
y *= x + 2;      等价于      y = y * (x + 2);
y &= x + 2;      等价于      y = y & (x + 2);
y <<= x + 2;     等价于      y = y << (x + 2);
```

复合赋值运算符的优先级和结合性与赋值运算符＝相同。

特别说明：Java 语言与 C/C++语言的区别(**运算符与表达式**)是，Java 语言没有无符号数的概念，都是有符号数。有符号数在右移时分带符号右移(>>)和不带符号右移(>>>)两种。

本节习题

1. Java 表达式"5+2.0"，该表达式结果的数据类型和值分别是(　　)。
   A. short,7　　B. int,7　　C. float,7.0　　D. double,7.0
2. Java 表达式"5 / 2"，该表达式结果的数据类型和值分别是(　　)。
   A. short,2　　B. int,2　　C. float,2.5　　D. double,2.5
3. Java 表达式"9 % 5"，该表达式结果的数据类型和值分别是(　　)。
   A. short,1　　B. int,4　　C. float,1.8　　D. double,4.0
4. 执行 Java 语句"int x=5,y; y=x++;"之后，变量 x 和 y 的值分别为(　　)。
   A. 5,5　　B. 5,6　　C. 6,5　　D. 6,6
5. 执行 Java 语句"int x=5,y; y=--x;"之后，变量 x 和 y 的值分别为(　　)。
   A. 4,4　　B. 4,5　　C. 5,4　　D. 5,5
6. 位与运算表达式"1001 & 0110"的结果是(　　)。
   A. 1001　　B. 0110　　C. 0000　　D. 1111

7. 位或运算表达式"1001|0110"的结果是(　　)。

A. 1001　　B. 0110　　C. 0000　　D. 1111

8. 异或运算表达式"1001 ^ 0110"的结果是(　　)。

A. 1001　　B. 0110　　C. 0000　　D. 1111

9. 执行 Java 语句"int x=5; double y=10.5; y-=x/2.0;"之后,变量 y 的值为(　　)。

A. 2.25　　B. 5.0　　C. 8.0　　D. 8.5

10. 执行 Java 语句"int x=5; double y=10.5; y/=x/2.5;"之后,变量 y 的值为(　　)。

A. 2.5　　B. 5.0　　C. 5.25　　D. 12.5

2.4 算法结构与控制语句

算法有 3 种基本结构,分别是顺序结构、选择结构和循环结构。按书写顺序依次执行操作步骤的算法称为**顺序结构**算法。顺序结构是最简单的一种算法结构。算法中,某些操作步骤需要满足特定条件才被执行,这种算法结构称为**选择结构**。还有一些算法,在满足特定条件下将重复执行某些操作步骤,这种算法结构称为**循环结构**。

选择结构和循环结构都要用到**条件**。如果一个条件成立,称这个条件为**真**(true),否则称之为**假**(false)。Java 语言使用**布尔类型**来表示条件的真假,通过**关系运算符**(例如大于、小于或等于)构成的关系表达式来描述一个条件,通过**逻辑运算符**(与、或、非)构成的逻辑表达式来描述一个复合条件。

使用 Java 语言将设计好的算法编写成一组语句序列,这就是 Java 源程序。为了描述选择结构和循环结构的算法,Java 语言分别提供了**选择语句**和**循环语句**。

2.4.1 布尔类型及其运算

Java 语言使用**布尔类型**(boolean)来表示条件的真假。布尔类型的取值只有两个,即 **true**(真)和 **false**(假)。boolean、true 和 false 都是 Java 语言的关键字。

1. 关系运算符

Java 语言提供 6 个关系运算符,用于比较两个数之间的大小。这 6 个关系运算符是>(大于)、>=(大于或等于)、<(小于)、<=(小于或等于)、==(等于)、!=(不等于)。

由关系运算符构成的表达式称为关系表达式,其运算结果是布尔类型。例 2-1 列举了一些关系表达式的例子。

例 2-1　关系表达式举例

关系表达式	布尔类型结果	备　注
5>3	true	5 大于 3 吗? 是的
5>=3	true	5 大于或等于 3 吗? 是的

续表

关系表达式	布尔类型结果	备　注
5<=3	false	5 小于或等于 3 吗？不是
5==3	false	5 等于 3 吗？不是
5!=3	true	5 不等于 3 吗？是的
2+3<=1+2	false	比较两个算术表达式时，先计算表达式，再比较其结果。算术运算符优先级高于关系运算符

选择结构或循环结构中的条件，通常用于判断程序中变量当前数值的大小。假设已定义变量 x：

```
int x = 10;
```

则例 2-2 中的关系表达式都可以构成一个条件。

例 2-2　由关系表达式所描述的条件举例(假设 int x=10;)

条　件	布尔类型结果	条件是否成立
x>5	true	x 大于 5 吗？是的，条件成立
x<5	false	x 小于 5 吗？不是，条件不成立
x-5==5	true	x-5 等于 5 吗？是的，条件成立
x-5<0	false	x-5 小于 0 吗？不是，条件不成立

2. 逻辑运算符

Java 语言提供 3 个逻辑运算符(见表 2-4)，用于将多个条件组合成一个复合条件。

表 2-4　逻辑运算符

逻辑运算符	运 算 规 则
&&(逻辑与)	双目运算符。若两个操作数都为 true，则结果为 true；否则为 false。相当于“**并且**”的意思
||(逻辑或)	双目运算符。若两个操作数中有一个为 true，则结果为 true；否则为 false。相当于“**或**”的意思
!(逻辑非)	单目运算符。若操作数为 true 则结果为 false；若操作数为 false 则结果为 true。相当于“**求反**”的意思

由逻辑运算符构成的表达式称为**逻辑表达式**，其运算结果是布尔类型。参与逻辑运算的操作数必须是布尔类型。假设已定义变量 x、y：

```
int x = 10, y = 20;
```

则例 2-3 中的逻辑表达式都可以构成一个复合条件。

例 2-3 由逻辑表达式所描述的复合条件举例(假设 int x=10,y=20;)

复合条件	布尔类型结果	复合条件是否成立
x>5 && y>10	true	条件成立
x<5 || y<10	false	条件不成立
x-5==5 || y==0	true	条件成立
!(x>5)	false	条件不成立

2.4.2 选择语句

有些算法,其中的某些操作步骤需满足特定条件才被执行。例如,给定 x 的值,求其倒数。当 x 等于 0 时,倒数 1/x 没有意义。因此在设计求倒数算法时,应当判断条件“x 不等于 0”是否成立。如果成立则求 x 的倒数,否则应提示错误信息。具体的求倒数算法见例 2-4。

例 2-4 算法举例:给定 x 的值,求其倒数

```
1  定义变量 x,申请保存数值的内存空间。
2  从键盘输入变量 x 的值。
3  如果条件"x 不等于 0"成立,则转到 4 计算倒数,否则转到 5 提示错误信息。
4  计算并显示表达式"1/x"的结果,转 6。
5  条件"x 不等于 0"不成立(即 x 等于 0),显示错误信息。
6  算法结束。
```

例 2-4 算法中的第 3~5 步使用的是一种自然语言里常用的句型,即“**如果……,就……,否则……**”。在算法设计中,这种句型描述的是“如果**条件**成立,则执行**算法分支 1**,否则执行**算法分支 2**”,这种类型的算法结构被称为**选择结构**或分支结构。条件、算法分支 1 和算法分支 2 是选择结构中的 3 个要素。

Java 语言提供了两种**选择语句**(decision-making statement)来描述选择结构的算法,分别是 if-else 语句和 switch-case 语句。

1. if-else 语句

Java 语法:**if-else 语句**

```
if (表达式)
{  语句 1  }
else
{  语句 2  }
```

语法说明:

- **表达式**指定一个判断条件。该表达式结果应为布尔类型,例如关系表达式或逻辑表达式。
- **语句 1** 是描述算法分支 1 的 Java 语句序列,即条件成立时执行的语句序列。
- **语句 2** 是描述算法分支 2 的 Java 语句序列,即条件不成立时执行的语句序列。如果条件不成立时不需要做什么处理,则省略 else 和{语句 2}。

- 语句1、语句2可能是包含多条Java语句的序列，此时必须用一对大括号将它们括起来。如果只包含一条语句，则大括号可以省略。
- 计算机执行该语句时，首先计算表达式(即判断条件)，若结果为true(条件成立)，则执行语句1；否则，执行else后面的语句2。

使用if-else语句将例2-4的求倒数算法编写成Java程序，见例2-5。

例2-5　实现求倒数算法的Java程序(if-else语句)

```
1   import java.util.Scanner;                        //导入外部程序 Scanner
2
3   public class JavaTest {                          //主类
4       public static void main(String[] args) {     //主方法
5           Scanner sc = new Scanner( System.in );//创建扫描器对象 sc
6           double x;                                //定义一个 double 型变量 x
7           x = sc.nextDouble();                     //从键盘输入变量 x 的值
8
9           if (x != 0) {                            //判断条件"x 不等于 0"是否成立
10              //条件成立时执行下列代码。因为是多条语句,所以用大括号括起来
11              double y;                    //再定义一个 double 型变量 y,用于保存 x 的倒数
12              y = 1 / x;                           //求 x 的倒数,结果赋值给 y
13              System.out.println( y );             //显示 y 的值,即 x 的倒数
14          }
15          else                                     //else 分支只有一条语句,可省略大括号
16              System.out.println( "0 的倒数没有意义" );     //显示错误信息
17      }
18  }
```

用一对大括号括起来的语句序列称为**复合语句**。例2-5中的第10～13行被一对大括号括起来，这就构成了一条复合语句。在语法上，Java语言将复合语句当作一条语句。有了复合语句，if-else语句的语法定义可以省略大括号，改写成如下形式：

```
if (表达式)
  语句 1
else
  语句 2
```

其中，语句1、语句2可以是单条语句，也可以是由大括号括起来的复合语句。Java语言还有一种特殊的**空语句**，即仅由分号“;”构成的语句。计算机执行空语句时不做任何处理。如无特别说明，本书后续语法定义中的术语“**语句**”都将包括复合语句和空语句。

例2-6给出一个判断年份是否是公历闰年的Java程序。平年的二月只有28天，而闰年的二月有29天。粗略地说是四年一闰，而准确判断闰年的条件是：年份能被4整除并且不能被100整除，或者年份能被400整除。该条件比较复杂，例2-6代码第9行通过取余运算和关系运算来描述“整除”条件，再通过逻辑运算来描述“并且”和“或者”这样的复合条件。

例 2-6 判断年份是否是闰年的 Java 程序

```
1  import java.util.Scanner;                          //导入外部程序 Scanner
2
3  public class JavaTest {                            //主类
4      public static void main(String[] args) {       //主方法
5          Scanner sc = new Scanner( System.in );//创建扫描器对象 sc
6          int year;                                  //定义一个 int 型变量 year
7          year = sc.nextInt();                       //从键盘输入一个年份,保存到变量 year 中
8
9          if ((year%4==0&&year%100!=0)||year%400==0)     //判断闰年条件是否成立
10             System.out.println( year + "是闰年" );      //条件成立则该年份是闰年
11         else
12             System.out.println( year + "不是闰年" );    //否则该年份不是闰年
13     }
14 }
```

例 2-7 给出了使用 if-else 语句求解符号函数的 Java 程序。符号函数的定义如下：

$$\operatorname{sgn}(x)=\begin{cases}1 & (x>0)\\ 0 & (x=0)\\ -1 & (x<0)\end{cases}$$

例 2-7 求符号函数 sgn(x)的 Java 程序

```
1  import java.util.Scanner;                          //导入外部程序 Scanner
2
3  public class JavaTest {                            //主类
4      public static void main(String[] args) {       //主方法
5          Scanner sc = new Scanner( System.in ); //创建扫描器对象 sc
6          float x;                                   //定义一个 float 型变量 x
7          x = sc.nextFloat();                        //从键盘输入变量 x 的值
8
9          int sgn;                     //定义一个 int 型变量 sgn,用于保存符号函数的结果
10         if (x == 0)                  //首先将 x 分为等于 0 和不等于 0 两种情况
11             sgn = 0;                 //x = 0 的情况
12         else {                       //在 x 不等于 0 时,再进一步区分 x > 0 和 x < 0 这两种情况
13             if (x > 0)  sgn = 1;     //x > 0 的情况
14             else  sgn = -1;          //x < 0 的情况
15         }
16         System.out.println( sgn ); //显示 sgn 的值,即符号函数的结果
17     }
18 }
```

例 2-7 中的代码第 13～14 行是在 if-else 语句中**嵌套**的另一个 if-else 语句。if-else 语句可以多层嵌套。多层嵌套时应注意：每个 else 自动和上面最近的 if 配对。如果 if-else 配对错误,执行程序得到的结果通常也是错误的。为保险起见,上层 if-else 语句应添加大括号将下层的 if-else 括起来,例如上述代码第 12 和 15 行的大括号。

编写 Java 程序时,良好的**书写格式**对程序的阅读理解非常有帮助。例如例 2-7 中的代

码第13、14行，大括号内部语句的**缩进**就是一种很好的书写格式。缩进可以体现语句的层次。添加**注释**、适当的空行或空格等也都是好的书写格式。

多条比较短的语句可以写在一行，一条长的语句也可以写成多行。程序的书写格式主要是为方便程序员阅读，不会影响程序语法的正确性。例如例2-7中的代码第10、11行可以写在同一行：

```
if (x == 0)  sign = 0;
```

例2-7中的求解符号函数算法实际上是一种**多分支结构**算法。描述多分支结构算法可以改用另一种特殊的if-else句型，即if-else if语句，如例2-8所示。

例2-8　求符号函数sgn(x)的Java程序(if-else if语句)

```
1   import java.util.Scanner;                      //导入外部程序 Scanner
2
3   public class JavaTest {                        //主类
4       public static void main(String[] args) {   //主方法
5           Scanner sc = new Scanner( System.in ); //创建扫描器对象 sc
6           float x;                               //定义一个 float 型变量 x
7           x = sc.nextFloat();                    //从键盘输入变量 x 的值
8
9           int sgn;                  //定义一个 int 型变量 sgn,用于保存符号函数的结果
10          if (x == 0)  sgn = 0;     //首先检查 x 等于 0 的情况
11          else if (x > 0)  sgn = 1; //再检查 x 大于 0 的情况
12          else  sgn = -1;           //最后剩下的就是 x 小于 0 的情况
13          System.out.println( sgn ); //显示符号函数的结果
14      }
15  }
```

Java语法：if-else if **语句**

```
if (表达式 1)           语句 1
else if (表达式 2)      语句 2
…
else if (表达式 n)      语句 n
else                    语句 n+1
```

语法说明：

- **表达式**1～n分别是需依次判断的条件。表达式结果应为布尔类型，例如关系表达式或逻辑表达式。
- **语句**1～n分别对应条件成立时执行的语句，可以是单条语句、复合语句或空语句。
- **语句**n+1是所有条件都不成立时执行的语句，可以是单条语句、复合语句。如果所有条件都不成立时不需要做什么处理，即空语句，则省略else和语句n+1。
- 计算机执行该语句时，首先计算表达式1，若为true则执行语句1；否则继续计算表达式2，……，直到表达式n；如果所有条件都不成立则执行else后面的语句n+1。计算机只会执行语句1～n+1中的一条。

if-else if 语句适用于描述多分支结构算法。例 2-9 给出了另一个应用 if-else if 语句的程序实例。该程序的功能是输入表示星期几的数值(1～7),然后显示其对应的英文单词。

例 2-9 显示星期几英文单词的 Java 程序

```
1  import java.util.Scanner;                          //导入外部程序 Scanner
2
3  public class JavaTest {                            //主类
4      public static void main(String[ ] args) {      //主方法
5          Scanner sc = new Scanner( System.in );     //创建扫描器对象 sc
6          int x;                                     //定义一个 int 型变量 x
7          x = sc.nextInt();      //从键盘输入一个表示星期几的数值(1～7),保存到变量 x 中
8
9          //下列 if - else if 语句根据 x 的值显示其对应的英文单词
10         if (x == 1)  System.out.println( "Monday" );
11         else if (x == 2)  System.out.println( "Tuesday" );
12         else if (x == 3)  System.out.println( "Wednesday" );
13         else if (x == 4)  System.out.println( "Thursday" );
14         else if (x == 5)  System.out.println( "Friday" );
15         else if (x == 6)  System.out.println( "Saturday" );
16         else if (x == 7)  System.out.println( "Sunday" );
17         else  System.out.println( "Input Error" );  //输入数值不在 1～7 内,提示错误
18     }
19 }
```

2. 条件运算符

下面再介绍一下 Java 语言中的**条件运算符**"? :"。在程序设计中,"如果**条件**成立,则执行**算法分支 1**,否则执行**算法分支 2**"是一种常用的算法结构。例如,比较两个变量 a、b 的大小,将其中较大的数赋值给 c,用 if-else 语句编写的示例代码如下:

```
int a = 5, b = 10, c;
if (a > b)  c = a;
else  c = b;
```

Java 语言提供了一种特殊的条件运算符,可以实现这样比较简单的 if-else 结构。

Java 语法:**条件运算符"? :"**

```
表达式 ? 表达式 1 : 表达式 2
```

语法说明:

- 条件运算符将 3 个表达式连接在一起,构成一个大的**条件表达式**。其中的**表达式**指定一个判断条件,该表达式结果应为布尔类型,例如关系表达式或逻辑表达式。
- 如果表达式的结果为 true,则计算**表达式 1**,将其结果作为整个条件表达式的结果;否则计算**表达式 2**,将其结果作为整个条件表达式的结果。
- 条件运算符为 3 目运算符。

举例：

```
int a = 5, b = 10, c;
a>b ? a : b                    //这是一个条件表达式,其结果等于 10,数据类型为 int 型
System.out.println( a>b ? a : b ); //显示条件表达式的结果
c = ( a>b ? a : b );           //将条件表达式的结果赋值给变量 c
```

3. switch-case 语句

多分支结构算法中有这样一类特殊的算法：某一表达式的结果可分为若干种情况，每种情况执行一个算法分支。例 2-10 具体描述了这样一类特殊的多分支结构算法。

例 2-10　一类特殊的多分支结构算法

```
1  计算某个表达式,判断其结果属于下列哪种情况。
2  情况 1:执行算法分支 1,执行结束转到 7。
3  情况 2:执行算法分支 2,执行结束转到 7。
4  …
5  情况 n:执行算法分支 n,执行结束转到 7。
6  否则属于其他情况:执行算法分支 n+1,执行结束转到 7。
7  算法结束。
```

Java 语言提供的 switch-case 语句可描述这类特殊的多分支结构算法。

Java 语法：switch-case **语句**

```
switch (表达式) {
case 常量表达式 1: 语句 1
case 常量表达式 2: 语句 2
…
case 常量表达式 n: 语句 n
default: 语句 n+1
}
```

语法说明：

- 计算机执行该语句时，首先计算 switch 后面的**表达式**，然后将结果依次与各 case 后的**常量表达式**的结果进行比对。若比对成功，则以比对成功的 case 语句为起点，顺序执行后面的所有语句，直到整个 switch-case 语句结束；或遇到 break 语句时中途跳出 switch-case 语句。如果所有比对都不成功，则将 default 语句作为执行的起点。
- **表达式**的结果应当是整型或字符型（即 byte、short、int、long 或 char 型），不能是浮点型。
- **常量表达式 1～n** 分别列出 switch 后面“表达式”可能的结果。常量表达式只能是常量，或由常量组成的表达式。各常量表达式的结果不能相同。
- **语句 1～n** 分别对应常量表达式比对成功时应执行的语句序列。通常都在末尾增加一条 break 语句，这样可以宣告算法结束，中途跳出。
- **语句 n+1** 是 default 后面的语句，即所有比对都不成功时应执行的语句。default 语

句习惯上被放在最后。语句 1～n+1 为复合语句时，大括号也可省略。

switch-case 语句俗称为**开关语句**。可以将例 2-9 显示星期几英文单词的程序改用 switch-case 语句来实现，见例 2-11。

例 2-11 显示星期几英文单词的 Java 程序(switch-case 语句)

```
1  import java.util.Scanner;                              //导入外部程序 Scanner
2
3  public class JavaTest {                                //主类
4      public static void main(String[] args) {           //主方法
5          Scanner sc = new Scanner( System.in );         //创建扫描器对象 sc
6          int x;                                         //定义一个 int 型变量 x
7          x = sc.nextInt();     //从键盘输入一个表示星期几的数值(1～7),保存到变量 x 中
8          //下列 switch-case 语句根据 x 的值显示对应的英文单词
9          switch ( x ) {
10         case 1:  System.out.println( "Monday" );   break;
11         case 2:  System.out.println( "Tuesday" );   break;
12         case 3:  System.out.println( "Wednesday" );   break;
13         case 4:  System.out.println( "Thursday" );   break;
14         case 5:  System.out.println( "Friday" );   break;
15         case 6:  System.out.println( "Saturday" );   break;
16         case 7:  System.out.println( "Sunday" );   break;
17         default:  System.out.println( "Input Error" );   break;
18         }
19         //每个 case 语句显示出对应的英文单词之后,程序功能即已完成
20         //因此使用 break 语句中途跳出 switch 语句
21 }  }
```

一年有 12 个月，月份有大小。大月份为 31 天，小月份为 30 天。例 2-12 的程序能够显示不同月份的天数。

例 2-12 显示不同月份天数的 Java 程序(switch-case 语句：共用语句)

```
1  import java.util.Scanner;                          //导入外部程序 Scanner
2
3  public class JavaTest {                            //主类
4      public static void main(String[] args) {       //主方法
5          Scanner sc = new Scanner( System.in );     //创建扫描器对象 sc
6          int month;                                 //定义一个 int 型变量 month
7          month = sc.nextInt();      //从键盘输入一个月份(1～12),保存到变量 month 中
8          //下列 switch-case 语句显示不同月份的天数
9          switch ( month ) {
10         case 1:                    //1 月大
11         case 3:                    //3 月大
12         case 5:                    //5 月大
13         case 7:                    //7 月大
14         case 8:                    //8 月大
15         case 10:                   //10 月大
16         case 12:  System.out.println( "31 天" );  break;  //1、3、5、7、8、10、12 月共用语句
17         case 4:                    //4 月小
```

```
18           case 6:                                              //6 月小
19           case 9:                                              //9 月小
20           case 11: System.out.println( "30 天" );        break;   //4、6、9、11 月共用语句
21           case 2:  System.out.println( "28 或 29 天" ); break;   //2 月
22           default: System.out.println( "Input Error" ); break;   //提示错误信息
23           }
24   }  }
```

例 2-12 中，所有的大月份都是 31 天，因此 case 1、3、5、7、8、10、12 所执行的输出语句应当是一样的。类似地，所有的小月份 case 4、6、9、11 也是相同的。switch-case 语句中，不同的 case 可以共用语句。

可以看出，case 比对的过程实际上是在查找 switch 语句执行的起点。查找到起点后，计算机将从该起点开始，顺序执行后面的所有语句，直到 break 语句中途跳出或整个 switch-case 语句结束为止。在 switch-case 语句中，break 语句的作用是中途跳出，转去执行 switch-case 语句后面的下一条语句。

2.4.3 循环语句

有一些算法，在满足特定条件下将重复执行某些操作步骤，这种算法结构称为**循环结构**。

例如一个奇数数列：1,3,5,7,9,…，如需计算数列前 N 项的累加和，该如何设计算法呢？通过数学方法，可以将这个求累加和问题描述为：$\sum_{n=1}^{N} 2n-1$。这种描述方法本身就体现了一种求解累加和的算法思想。

首先，引入一个表示数列项的变量 n，第 n 项可表示为 $2n-1$。求解前 N 项累加和的过程是一个重复累加的过程。累加起点是 $n=1$，累加条件是 $n\leqslant N$。每次累加所做的操作就是累加第 n 项（当前项）的值 $2n-1$，然后将 n 加 1，准备下一次累加。重复该累加操作，直到累加条件 $n\leqslant N$ 不成立。

上述求解累加和的算法就是一种循环结构算法。每次循环后，算法所引入的变量 n 都会发生变化（加 1），然后通过比较 n 的值来控制是否继续循环，即检查条件 $n\leqslant N$ 是否成立。像变量 n 这样用于控制循环次数的变量被称为**循环变量**。

循环结构用自然语言描述就是这样一种句型，即：“**如果……，就重复做……，否则停止**”。在算法设计中，这种句型描述的是“如果**条件**成立，则重复执行**循环体**，否则**结束循环**”。一个循环结构由 4 个要素构成，它们分别是：**循环变量**、循环变量的**初始值**、**循环条件**和**循环体**。例 3-17 具体描述了上述求解累加和的循环结构算法。

例 2-13 算法举例：求解奇数数列前 N 项累加和的循环结构算法

```
1  首先定义一个 int 型变量 N,从键盘输入 N 的值。
2  定义一个循环变量 n(初始值为 1),表示当前数列项的序号。
3  再定义一个 int 型变量 sum(初始值为 0),用于保存累加的结果。
4  开始循环:如果循环条件 n≤N 成立,则转到 5 做累加操作,否则转到 7 结束循环。
5  将当前项的值 2n - 1 累加到 sum 上:sum += 2n - 1。
6  将 n 加 1,准备下一次累加,转到 4 继续循环。步骤 5～6 是被重复执行的循环体。
7  循环结束后,显示 sum 的值,此时 sum 中的值就是数列前 N 项的累加和。
8  算法结束。
```

Java 语言提供了 3 种**循环语句**(looping statement)来描述循环结构的算法，它们分别是 **while** 语句、**do-while** 语句和 **for** 语句。

1. while 语句

Java 语法：while **语句**

```
while (表达式)
    语句
```

语法说明：

- **表达式**指定一个循环条件。该表达式结果必须是布尔类型，例如关系表达式或逻辑表达式。
- **语句**是描述循环体的 Java 语句，即条件成立时循环执行的算法。如循环条件一开始就不成立，则循环体一次也不执行。循环体中应包含使循环条件趋向于 false 的语句，否则循环条件一直为 true，循环体将无休止地执行，俗称为**死循环**。
- 计算机执行该语句时，首先计算表达式(即循环条件)，若结果为 true(条件成立)，则重复执行循环体语句；否则结束循环。

使用 while 语句将例 2-13 的求累加和算法编写成 Java 程序，见例 2-14。

例 2-14 求解奇数数列前 *N* 项累加和的 Java 程序(while 语句)

```
1  import java.util.Scanner;                     //导入外部程序 Scanner
2
3  public class JavaTest {                       //主类
4      public static void main(String[] args) {  //主方法
5          Scanner sc = new Scanner( System.in );//创建扫描器对象 sc
6          int N;                                //定义一个 int 型变量 N
7          N = sc.nextInt();                     //从键盘输入变量 N 的值
8
9          int n = 1, sum = 0;                   //定义循环变量 n(初始值为 1)
10                                               //定义保存累加结果的变量 sum(初始值为 0)
11         while (n <= N) {                      //用小括号将循环条件 n <= N 括起来
12             sum += 2 * n - 1;                 //将当前项的值 2n - 1 累加到 sum 上
13             n++;    //将 n 加 1,准备下一次累加。该语句使得循环条件 n <= N 趋向于 false
14             //执行完循环体最后一条语句之后,转到第 11 行,重新判断循环条件
15         }
16         //如果循环条件不成立,则循环结束,继续执行 while 语句的下一条语句
17         System.out.println( sum );            //显示变量 sum 的值,即前 N 项的累加和
18     }
19 }
```

2. do-while 语句

while 语句所描述的循环结构是“如果**条件**成立，则重复执行**循环体**，否则**结束循环**”。该循环结构的一个变形是“**先执行**循环体，**再判断条件**。如果条件成立则重复执行循环体，

否则结束循环”。Java 语言使用 do-while 语句来描述这种循环结构。

Java 语法：do-while **语句**

```
do {
    语句
} while (表达式) ;
```

语法说明：

- **表达式**指定一个循环条件。将条件放在循环体语句的后面，即先执行循环体，再判断条件。该表达式结果必须是布尔类型，例如关系表达式或逻辑表达式。
- **语句**是描述循环体的 Java 语句，不管循环条件是否成立，循环体至少执行一次。如果循环体只包含一条语句，则大括号“{}”可以省略。循环体中应包含使循环条件趋向于 false 的语句，否则将造成死循环。
- 计算机执行该语句时，首先执行一次循环体，然后再计算表达式(即循环条件)，若结果为 true(条件成立)，则重复执行循环体语句；否则结束循环。

使用 do-while 语句也可以实现例 2-13 的求累加和算法，见例 2-15。

例 2-15　求解奇数数列前 N 项累加和的 Java 程序(do-while 语句)

```
1   import java.util.Scanner;                          //导入外部程序 Scanner
2
3   public class JavaTest {                            //主类
4       public static void main(String[] args) {       //主方法
5           Scanner sc = new Scanner( System.in );     //创建扫描器对象 sc
6           int N;                                     //定义一个 int 型变量 N
7           N = sc.nextInt();                          //从键盘输入变量 N 的值
8
9           int n = 1, sum = 0;                      //定义循环变量 n(初始值为 1)
10                                                   //定义保存累加结果的变量 sum(初始值为 0)
11          do {                                     //先执行循环体
12              sum += 2 * n - 1;                    //将当前项的值 2n - 1 累加到 sum 上
13              n++;                                 //将 n 加 1,准备下一次累加
14          } while (n <= N) ;     //后判断条件。如条件成立则重复执行循环体,否则结束循环
15          //循环结束后,继续执行 do - while 语句的下一条语句
16          System.out.println( sum );               //显示变量 sum 的值,即前 N 项的累加和
17      }
18  }
```

while 语句是先判断条件，再决定是否执行循环体，循环体可能一次也不执行。而 do-while 语句是先执行循环体，再判断条件，循环体至少执行一次。通常情况下，这个细微的差别对算法结果没有影响，两种语句可以互相替换使用。但如果一开始的初始条件就不成立，则 while 语句和 do-while 语句的执行结果会有差异。例如，在求累加和的程序中，如果从键盘输入的 N 是 0，例 2-14 使用 while 语句的计算结果正确的(0，没有累加)，而例 2-15 使用 do-while 语句的计算结果是错误的(1，累加了一次)。

3. for 语句

Java 语言中,循环结构"如果**条件**成立,则重复执行**循环体**,否则**结束循环**"还可以用 for 语句描述。使用 for 语句来描述循环结构算法,形式更加紧凑。

Java 语法: for **语句**

```
for (表达式 1; 表达式 2; 表达式 3)
    语句
```

语法说明:

- **表达式 1** 只在正式循环前执行一次,通常用于为循环算法赋初始值。
- **表达式 2** 指定一个循环条件。每次循环时,先计算该表达式,如果为 true 则执行下面的循环体语句,否则结束循环。
- **表达式 3** 在每次循环体执行结束之后都被执行一次,主要用于修改循环条件中的某些变量,使循环条件趋向于 false。
- **语句**是描述循环体的 Java 语句。
- 计算机执行该语句时,首先计算表达式 1(通常为赋初始值);再计算表达式 2(即循环条件),若结果为 true 则重复执行循环体语句,每次执行完循环体语句之后都计算一次表达式 3(通常用于修改循环条件中的某些变量),然后再返回表达式 2 重新判断条件;若表达式 2 的结果为 false 则结束循环。

for 语句是 3 种循环语句中最简洁的语句。使用 for 语句替换例 2-14 中的 while 语句,同样可以实现求累加和的算法,见例 2-16。

例 2-16 求解奇数数列前 *N* 项累加和的 Java 程序(for 语句)

```
1   import java.util.Scanner;                           //导入外部程序 Scanner
2
3   public class JavaTest {                             //主类
4       public static void main(String[ ] args) {       //主方法
5           Scanner sc = new Scanner( System.in );      //创建扫描器对象 sc
6           int N;                                      //定义一个 int 型变量 N
7           N = sc.nextInt();                           //从键盘输入变量 N 的值
8
9           int n, sum = 0;             //定义循环变量 n
10                                      //定义保存累加结果的变量 sum(初始值为 0)
11          for (n = 1; n <= N; n++) {  //for 语句集中用 3 个表达式指定 n 的初始值 1、循环
12                              //条件 n <= N 以及修改循环变量 n++,使循环条件趋向于 false
13              sum += 2 * n - 1;       //循环体被简化了,原来的 n++语句被放入到 for 语句里面
14          }                           //循环体只有一条语句,此时这对大括号可以省略
15          System.out.println( sum );  //显示变量 sum 的值,即前 N 项的累加和
16      }
17  }
```

4. break 语句和 continue 语句

计算机通常是按照语句的书写顺序来执行 Java 程序的，而某些语句会造成执行顺序的跳转。例如下面的示例代码：

```
int a = 5, b = 10, c;          //将 a、b 中较大的数赋值给 c
if (a > b)
    c = a;
else
    c = b;
System.out.println( c );
```

其中，选择语句"if(a>b)"会造成执行顺序的跳转。当计算机执行该选择语句时，如果条件不成立则跳过语句"c=a;"，转去执行 else 后面的语句"c=b;"。这时，程序没有按照书写顺序执行，而是出现了跳转。

造成程序执行顺序跳转的语句被统称为**控制语句**。选择语句和循环语句都属于控制语句。下面再介绍另外两个常用的控制语句。

1）break 语句

我们已经知道，用 break 语句能够中途跳出 switch 语句，转去执行 switch 语句后面的下一条语句。使用 break 语句也能够中途跳出循环，转去执行该循环语句后面的下一条语句。例 2-17 是一个求圆面积的 Java 程序。

例 2-17　一个计算圆面积的 Java 程序

```
1  import java.util.Scanner;                        //导入外部程序 Scanner
2
3  public class JavaTest {                          //主类
4      public static void main(String[] args) {     //主方法
5          Scanner sc = new Scanner( System.in );   //创建扫描器对象 sc
6          double r;                                //定义一个变量 r 来存放圆的半径
7
8          r = sc.nextDouble();                     //从键盘输入圆的半径
9          System.out.println( 3.14 * r * r );      //显示圆面积
10     }
11 }
```

该程序一次只能求一个圆的面积，计算完就退出了。如果需要计算多个圆的面积，但不希望每次都重新启动程序，该怎么设计算法呢？答案是引入循环结构。

使用循环结构可以重复输入半径和显示圆面积的过程。但循环多少次应当由用户决定，该如何设定循环条件呢？可以将循环条件先设定为 true（即死循环），然后根据用户从键盘输入的半径来决定是否结束循环。如果用户输入的半径为正数，则计算圆面积，否则使用 break 语句跳出循环，程序结束。因为半径为 0 或负数是没有意义的，可用作结束循环的条件。具体程序代码如例 2-18 所示。

例 2-18 一个计算圆面积的 Java 程序(break 语句应用示例)

```
1   import java.util.Scanner;                        //导入外部程序 Scanner
2
3   public class JavaTest {                          //主类
4       public static void main(String[] args) {     //主方法
5           Scanner sc = new Scanner( System.in );   //创建扫描器对象 sc
6           double r;                                //定义一个变量 r 来存放圆的半径
7
8           while (true ) {                          //死循环
9               r = sc.nextDouble();                 //从键盘输入圆的半径
10              if (r <= 0)    break;          //如果用户输入的半径小于或等于 0,则跳出循环
11              System.out.println( 3.14 * r * r );  //计算并显示圆面积
12          }
13          //使用 break 语句中途跳出 while 语句,继续执行 while 语句的下一条语句
14      }
15  }
```

例 2-18 代码第 10 行是在循环语句中嵌套的一个 if 语句。Java 语言中,所有的选择语句和循环语句之间都可以互相嵌套。循环语句的相互嵌套,即一个循环语句中再包含另一个循环语句,被称为是**多重循环**。例 2-19 给出一个生成乘法表的 Java 程序,其中使用了两重循环。程序运行结果如图 2-1 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> JavaTest [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe (2017年12月23日 上午10:31:36)
1×1=1
1×2=2 2×2=4
1×3=3 2×3=6 3×3=9
1×4=4 2×4=8 3×4=12 4×4=16
1×5=5 2×5=10 3×5=15 4×5=20 5×5=25
1×6=6 2×6=12 3×6=18 4×6=24 5×6=30 6×6=36
1×7=7 2×7=14 3×7=21 4×7=28 5×7=35 6×7=42 7×7=49
1×8=8 2×8=16 3×8=24 4×8=32 5×8=40 6×8=48 7×8=56 8×8=64
1×9=9 2×9=18 3×9=27 4×9=36 5×9=45 6×9=54 7×9=63 8×9=72 9×9=81
```

图 2-1 运行例 2-19 程序所生成的乘法表

例 2-19 生成乘法表的 Java 程序 (多重循环应用示例)

```
1   public class JavaTest {                       //主类
2
3       public static void main(String[] args) {  //主方法
4           int x, y;                             //定义两个循环变量 x 和 y
5           for (x = 1; x <= 9; x++)  {           //第一重循环,x 从 1 到 9,共 9 行
6               for (y = 1; y <= x; y++)          //第二重循环,y 从 1 到 x。第 x 行有 x 个乘法
7                   System.out.print( y + "×" + x + "=" + (x * y) + " " );
8               System.out.print( '\n' );         //换一行,再显示后续的内容
9           }
10  }  }
```

注意：在多重循环中使用 break 语句，只能跳出它所在的本层循环。break 语句只能在 switch-case 语句和循环语句中使用，否则编译器会提示语法错误。

2）continue 语句

continue 语句的作用是结束本次循环，中途返回，继续下一次循环。例 2-20 给出一个 continue 语句的应用实例。

例 2-20　显示 1～50 所有能被 3 整除的数（continue 语句应用示例）

```
1  public class JavaTest {                          //主类
2
3      public static void main(String[] args) {  //主方法
4          for (int n = 1; n <= 50; n++)  {      //1～50 的循环
5              if (n % 3 != 0)   continue;       //如果 n 不能被 3 整除，则执行 continue 语句
6              //continue 语句的作用是结束本次循环，中途返回，去检查下一个数
7              //未中途返回的数是能被 3 整除的数，下面将显示这些数并用逗号隔开
8              System.out.print( n + ", " );
9          }
10 }  }
```

在循环语句中，continue 语句与 break 语句的区别是：continue 语句只结束本次循环，而 break 语句结束的是整个循环。另外，continue 语句只能在循环语句中使用，break 语句还可以在 switch-case 语句中使用。

3）带标号的 break 和 continue 语句

Java 语言中的 break 和 continue 语句可以借助标号直接跳出外层循环，或中途返回到外层循环。下面以 for 语句为例，给出一个带标号 break 和 continue 语句的代码结构。

```
Label1: for ( … ) {          //为外层循环语句添加标号，假设为 Label1
Label2:    for ( … ) {       //为内层循环语句添加标号，假设为 Label2
              …
             break Label1; //或 continue  Label1;
           }
         }
```

带标号的 break 语句可直接跳出标号指定的循环，多层循环时可以是任意的外层循环。同样，带标号的 continue 语句可直接返回标号指定的循环，多层循环时可以是任意的外层循环。例 2-21 给出一个使用带标号 continue 语句的示例程序。

例 2-21　显示 100～200 的所有质数（带标号的 continue 语句应用示例）

```
1  public class JavaTest {                                 //主类
2      public static void main(String[] args) {          //主方法
3          int i, j, n = 0;
4
5          Loop1: for (i = 101; i <= 200; i += 2) {     //外层循环，语句块标号 Loop1
6              Loop2: for(j = 2; j <= i/2; j++) {      //内层循环，语句块标号 Loop2
7                  if ( i % j == 0 )                    //不是质数，则中途返回
8                      continue Loop1;                  //借助标号 Loop1，直接返回外层循环
9              }
10             System.out.print( " " + i );             //是质数：显示质数，以空格隔开
11             n++;                                     //统计显示的质数个数，一行显示 10 个
```

```
12              if ( n < 10 )    //未满 10 个,则不换行
13                  continue;    //中途返回。无标号时直接返回本层循环,此处也为外层循环
14              System.out.println(); n = 0;     //换行显示,并将计数清零
15          }
16      }
17  }
```

注:带标号的 break 和 continue 语句类似于 C 语言里的 goto 语句,虽然可以灵活跳转,但不易阅读理解。作者不推荐使用这样的带标号的 break 和 continue 语句。

特别说明:Java 语言与 C/C++语言的区别(**控制语句**)如下。

- Java 语言选择语句和循环语句里的条件表达式必须是布尔类型,不能是其他类型。**注**:C/C++语言可以自动将其他类型转为布尔类型,将非 0 值转为 true,0 转为 false。
- Java 语言里的 break 和 continue 语句可以带标号,直接跳出外层循环,或直接返回外层循环。**注**:C/C++语言没有带标号的 break 和 continue 语句。

本节习题

1. 算法有 3 种基本结构,其中不包括(　　)。

   A. 顺序结构　　B. 并列结构　　C. 选择结构　　D. 循环结构

2. 算法结构(　　)不会用到条件。

   A. 顺序结构　　B. 选择结构

   C. 循环结构　　D. 以上 3 种都不需要

3. Java 表达式"5 <=5",该表达式结果的数据类型和值分别是(　　)。

   A. int,0　　B. int,true

   C. boolean,true　　D. boolean,false

4. 比较变量 x 的值是否等于 5,表达式(　　)的写法是正确的。

   A. x=5　　B. x==5　　C. x<>5　　D. x!=5

5. Java 表达式"1 >=0 && 0 <=1"的结果是(　　)。

   A. 0　　B. 1　　C. true　　D. false

6. 下列表达式中,(　　)的结果为 true。

   A. !(5 > 1)　　B. 5 > 1 && false

   C. 5 > 1 || false　　D. 5 < 1 || false

7. 执行 Java 语句"if(1 < 0 || false) System.out.print("Hello world!");"显示器上将显示(　　)。

   A. "Hello world!"　　B. Hello,world!

   C. Hello world!　　D. 什么都没显示

8. 执行下列 Java 语句:

```
int x = 15;
if (x % 2 == 0)   System.out.print( x/2 );
```

```
else  System.out.print( x/2 + 1 );
```

显示器上将显示(　　)。

A. 7　　B. 7.5　　C. 8　　D. 8.5

9. 执行下列 Java 语句：

```
int x = 2;
switch ( x ) {
case 1:  System.out.print("One" );  break;
case 2:  System.out.print("Two" );  break;
case 3:  System.out.print("Three" );  break;
default:  System.out.print("Error" );  break;
}
```

显示器上将显示(　　)。

A. One　　B. Two　　C. Three　　D. Error

10. 执行下列 Java 语句：

```
int x = 1;
switch ( x+1 ) {
case 1:  System.out.print("One" );
case 2:  System.out.print("Two" );
case 3:  System.out.print("Three" );
default:  System.out.print("Error" );
}
```

显示器上将显示(　　)。

A. One　　B. Two

C. TwoThree　　D. TwoThreeError

11. 执行下列 Java 语句：

```
int x = 5, y = 0;
while (x > 0) {
    y += 2;  x--;
}
```

执行结束后，x 和 y 的值分别为(　　)。

A. 5,0　　B. 0,5　　C. 5,10　　D. 0,10

12. 执行下列 Java 语句：

```
int x = 5, y = 0;
do {
    y += 2;  x--;
} while (x > 0);
```

执行结束后，x 和 y 的值分别为(　　)。

A. 5,0　　B. 0,5　　C. 5,10　　D. 0,10

13. 执行下列 Java 语句：

```
int x = 0, y = 0;
for (x = 5; x > 0; x-- )
    y += 2;
```

执行结束后，x 和 y 的值分别为(　　)。

A. 5,0　　B. 0,5　　C. 5,10　　D. 0,10

14. 执行下列 Java 语句：

```
int x = 5, y = 0;
while (x > 0) {
    y += 2;   x-- ;
    if (x % 4 == 0)   break;
}
```

执行结束后，x 和 y 的值分别为(　　)。

A. 5,0　　B. 0,10　　C. 4,2　　D. 4,4

15. 执行下列 Java 语句：

```
int x = 0;
while (x < 3)
    System.out.println(" * ");   x++;
```

显示器将显示(　　)。

A. 一个星号　　B. 两个星号

C. 三个星号　　D. 持续显示星号

本章学习要点

- Java 语言的基础语法大量借鉴了 C/C++语言。具有 C/C++语言基础的读者在学习 Java 语言时，只需重点了解它与 C/C++语言之间的区别。
- 本章应尽快熟悉 Java 语言编程环境。建议具有 C/C++语言基础的读者把之前学习过的 C/C++程序改用 Java 语言重写一遍。
- 通过对比可以知道，程序设计语言虽然语法不同，但设计思想是相同的。

本章习题

1. 编写程序。请编写一个计算表达式 x^3+2x^2+5 的 Java 程序。

2. 编写程序。分别用 while 语句、do-while 语句和 for 语句编写一个求阶乘 $N!$ 的 Java 程序。

3. 编写程序。我国古代《张丘建算经》中有这样一道著名的百鸡问题："鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一。百钱买百鸡，问鸡翁、母、雏各几何？"这道题的意思是：公鸡每只 5 元，母鸡每只 3 元，小鸡 3 只 1 元。用 100 元买 100 只鸡，问公鸡、母鸡和小

鸡各能买多少只？编写一个求解百鸡问题的 Java 程序。

4. 编写程序。求逆序数：从键盘任意输入一个 3 位整数，编程计算并输出它的逆序数。例如，输入－123，则输出－321。以下为程序的一个运行示例：

```
Input x: -123↙
y = -321
```

5. 编写程序。请自己提出一个数值计算问题，然后编写 Java 程序进行计算。

第2部分

面向对象程序设计方法

第3章 面向对象程序设计之一

程序的功能是数据处理，其中包括**数据**和**算法**两大部分。数据是程序处理的对象，对应程序中的变量或常量。算法是描述数据处理过程的一组操作步骤，这就是程序中的语句序列。大型程序的功能很强，这意味着要处理大量的数据，数据处理的算法也很多、很复杂。如何编写大型计算机程序呢？这就需要程序员学习**程序设计方法**。

程序设计方法的基本思想是：将大型程序中的数据和算法分解成**程序零件**，将不同零件的设计任务交由不同的程序员完成，这样就能以团队的形式来共同开发，然后将开发好的零件**组装**在一起，最终完成复杂的程序功能。目前，程序设计方法分为**结构化程序设计**和**面向对象程序设计**两种，它们分别采用不同的方式来分解和组装程序零件。

更进一步，如果所分解出的程序零件在以前项目中曾经开发过，或者可以从市场上购买到，那么就可以直接使用这些零件来组装软件，实现快速开发。使用已有的程序零件，实际上是重用其程序代码，这就是程序设计中的**代码重用**(code reuse)。为了让不同程序员开发的程序零件能够正确地组装在一起，在编写时应遵守共同的语法规则。因为易于复制，代码重用的成本很低，这是软件行业所独有的特点。代码重用可以极大地提高软件开发效率，代码重用也因此成为软件技术不断进步的主要动力。

为了应用程序设计方法来编写大型复杂程序，计算机语言需要提供描述和组装程序零件的语法规则。支持结构化程序设计方法的语言被称为结构化程序设计语言，支持面向对象程序设计方法的语言被称为面向对象程序设计语言。C 语言是一种结构化程序设计语言，Java 语言是一种面向对象程序设计语言。

本章将简单介绍结构化程序设计是如何演变到面向对象程序设计的，然后重点学习面向对象程序设计方法。

3.1 面向对象程序设计方法概述

程序是用于处理数据的，通常应包括如下 4 项功能。

(1) 定义保存数据的变量。

(2) 输入原始数据。

(3) 处理数据。

(4) 输出处理结果。

其中，(2)和(4)所完成的输入输出功能是程序提供给用户的交互界面，简称为**用户界面**。

本节通过一个程序实例,直观地介绍结构化程序设计是如何演变到面向对象程序设计的,并在程序的演变过程中具体讲解什么是面向对象的程序设计方法。

程序实例:编写一个计算长方形面积和周长的演示程序。假设程序由甲、乙两位程序员分工协作,共同编写。

因为 C++语言既支持结构化程序设计,又支持面向对象程序设计,因此下面的演示程序一开始会先使用 C++语言,然后再过渡到 Java 语言。

3.1.1 结构化程序设计中的函数

结构化程序设计方法就是将一个大型程序中的复杂算法分解成多个简单的模块,分而治之,然后将这些模块组装起来,最终形成一个完整的数据处理流程。C/C++语言支持结构化程序设计方法,以函数的语法形式来描述和组装模块,这就是函数的定义和调用。

编写计算长方形面积和周长的程序,可以使用结构化程序设计方法将程序划分成 3 个函数。两位程序员分工协作,甲负责编写主函数,乙负责编写计算面积和周长的子函数。下面先给出 C/C++语言中函数的定义和调用语法。

C/C++语法:**定义函数**

```
函数类型  函数名(形式参数列表)
{
    函数体
}
```

语法说明:

- **函数类型**指定函数返回值(即函数值)的数据类型。函数类型由函数功能决定,可以是除数组之外的任何数据类型,省略时默认为 int 型。某些函数可能只是完成某种功能,但没有返回值,此时函数类型应定义为 void。
- **函数名**指定函数的名称,由程序员命名,需符合标识符的命名规则。通常函数之间不能重名。
- **形式参数列表**定义了函数接收输入参数所需的变量,这些变量称为**形式参数**,简称为**形参**。可以有多个形参,每个形参以"**数据类型 变量名**"的形式定义,形参之间用逗号","隔开。某些函数可能不需要输入参数,此时形式参数列表省略为空。
- **函数体**是描述数据处理算法的 C/C++语句序列,用大括号"{ }"括起来。函数体中可以定义专供本函数使用的局部变量。如果函数有返回值,则应使用 return 语句返回,返回值的数据类型应与函数类型一致。
- "函数类型 函数名(形式参数列表)"是函数的头部,被称为**函数原型**(prototype)或**函数签名**(signature)。它定义了函数的调用接口,即函数名、输入参数和返回值类型。

C/C++语法:**调用函数**

```
函数名(实际参数列表)
```

语法说明：

- **函数名**指定被调用函数的名称。
- **实际参数列表**给出函数所需要的输入参数。调用函数时应按被调用函数的要求给定具体的输入参数值，这些参数值称为**实际参数**，简称为**实参**。实际参数可以是常量、变量或表达式，参数之间用逗号“,”隔开。调用时，计算机首先将实参值按位置顺序一一赋值给对应的形参变量，这称为函数调用时的**参数传递**。实参与形参应当个数一致，类型一致。
- “函数名(实际参数列表)”就是在**调用**某个函数。有返回值的函数调用可作为操作数参与表达式运算，该操作数等于函数定义里的返回值。某些函数可能只是完成某种功能，但没有返回值。无返回值的函数调用直接加分号“;”即构成一条完整的函数调用语句。
- 一个函数调用另一个函数，前一个函数称为**主调函数**，后面被调用的函数称为**被调函数**。
- 调用函数前，需要编写声明语句对被调函数进行**声明**。
- 函数声明语句就是函数原型加上分号，即“函数类型 函数名(形式参数列表);”。将多条函数声明语句集中放在某个**头文件**(.h)中，然后使用“#include”指令插入头文件，这样可以简化函数声明。

例3-1分别给出甲、乙两位程序员编写的计算长方形面积和周长的C++程序代码。

例3-1 计算长方形面积和周长的C++程序代码(**函数**)

程序员甲：主函数(1.cpp)

```
1  #include <iostream>           //C++语言的头文件
2  using namespace std;          //声明命名空间
3  //C语言：#include <stdio.h>
4  #include "2.h"                //插入头文件2.h
5
6  int main()
7  {
8      int a, b;              //定义保存长宽数据的变量
9      cin >> a >> b;            //输入长方形的长宽
10     //C语言：scanf("%d %d", &a, &b);
11
12     cout << Area(a, b) << endl;//长方形面积
13     cout << Len(a, b) << endl;//长方形周长
14     //C语言：printf("%d\n", Area(a, b));
15     //C语言：printf("%d\n", Len(a, b));
16     return 0;
17 }
```

程序员乙：子函数(2.cpp)

```
//计算长方形面积和周长的函数
int Area(int length, int width)
{
    return ( length * width );
}
int Len(int length, int width)
{
    return ( 2 * (length + width) );
}
```

程序员乙：头文件(2.h)

```
//声明外部函数的原型
int Area(int length, int width);
int Len(int length, int width);
```

例3-1程序的代码说明如下。

(1) 程序员甲负责编写主函数main()。

主函数中，程序员甲定义了两个保存长方形长、宽数据的变量a、b，通过cin指令接收用户输入的长、宽值，然后调用子函数Area()和Len()分别求出长方形的面积和周长，并通过

cout 指令将计算结果反馈给用户。其中的 cin 和 cout 指令为用户提供了一种命令行风格的交互界面。

程序员甲编写的主函数代码共完成了 4 项程序功能中的 3 项，即定义保存数据的变量、输入原始数据和输出处理结果。剩余的一项功能（即数据处理）是由程序员乙编写子函数代码完成的。程序员甲在计算长方形的面积和周长时通过调用子函数就轻松完成了数据处理功能。换句话说，程序员乙帮程序员甲分担了部分编程工作。

（2）程序员乙负责编写子函数。

程序员乙负责编写两个子函数 Area()和 Len()。子函数 Area()的功能是计算长方形的面积，子函数 Len()的功能是计算长方形的周长。程序员乙编写子函数时定义了两个形参 length 和 width，用于接收主函数传递过来的长、宽值（即存放在实参 a、b 中的值），然后编写算法代码求出长方形的面积或周长，并以返回值的形式将结果返回给主函数。

程序员乙将计算长方形面积或周长的算法代码定义成函数，程序员甲调用该函数就能实现相应的程序功能。调用函数实际上是重用该函数的代码，实现其规定的程序功能。程序员乙编写好的函数 Area()和 Len()可以在本次项目中使用，也可以在今后的项目中使用，或提供给任何其他程序员使用。不管是什么项目，或是哪位程序员，只要调用这两个函数就都能轻松实现求长方形面积或周长的功能。将算法代码定义成函数的好处是**“一次编写，长期使用”**。

是的，也许有人会觉得重用这两个求长方形面积或周长的算法代码没有什么价值，自己也能编写。但设想一下，如果这是一个 JPEG 图像压缩算法呢？本例中，求长方形的面积或周长仅仅是一个例子，函数才是读者应该关注的重点。

在结构化程序设计方法中，**函数**是重用算法代码的基本语法形式。

（3）两类不同的程序员角色。

代码重用可以减少开发工作量，提高软件质量。代码重用过程中，程序员有两类角色：一类是提供代码的程序员（**代码提供者**）；另一类是使用代码的程序员（**代码使用者**）。在例 3-1 的程序中，计算长方形面积和周长的子函数 Area()、Len()是被重用的代码。程序员乙编写重用代码，是代码提供者。程序员甲编写主函数时调用这两个子函数，他是代码使用者。**注**：名词“重用代码”指的是“被重用的代码”。

实际应用中，重用代码经常是由一位程序员编写，然后被多个程序员使用。也就是说，类似例 3-1 程序中程序员乙的角色只有一个人，而程序员甲的角色则有很多人（见图 3-1）。

如何让重用代码完成更多的程序功能，这是软件技术不懈追求的目标。如果程序员乙编写的重用代码能多完成一项功能，那么众多使用代码的程序员就都能少承担一项功能，这就从整体上提高了软件开发的效率。请注意，增加重用代码的功能，可以减少代码使用者的工作量，但反过来会增加代码提供者的工作量。编写重用代码的程序员应当具备“辛苦我一个，幸福千万家”的精神。

在例 3-1 中，程序员甲承担了 4 项程序功能中的 3 项（即定义保存数据的变量、输入原始数据和输出处理结果），而程序员乙只承担了一项数据处理功能。下面我们将基于这个例子来演示，如何将程序员甲当前所承担的 3 项程序功能一步一步地继续转移给程序员乙。

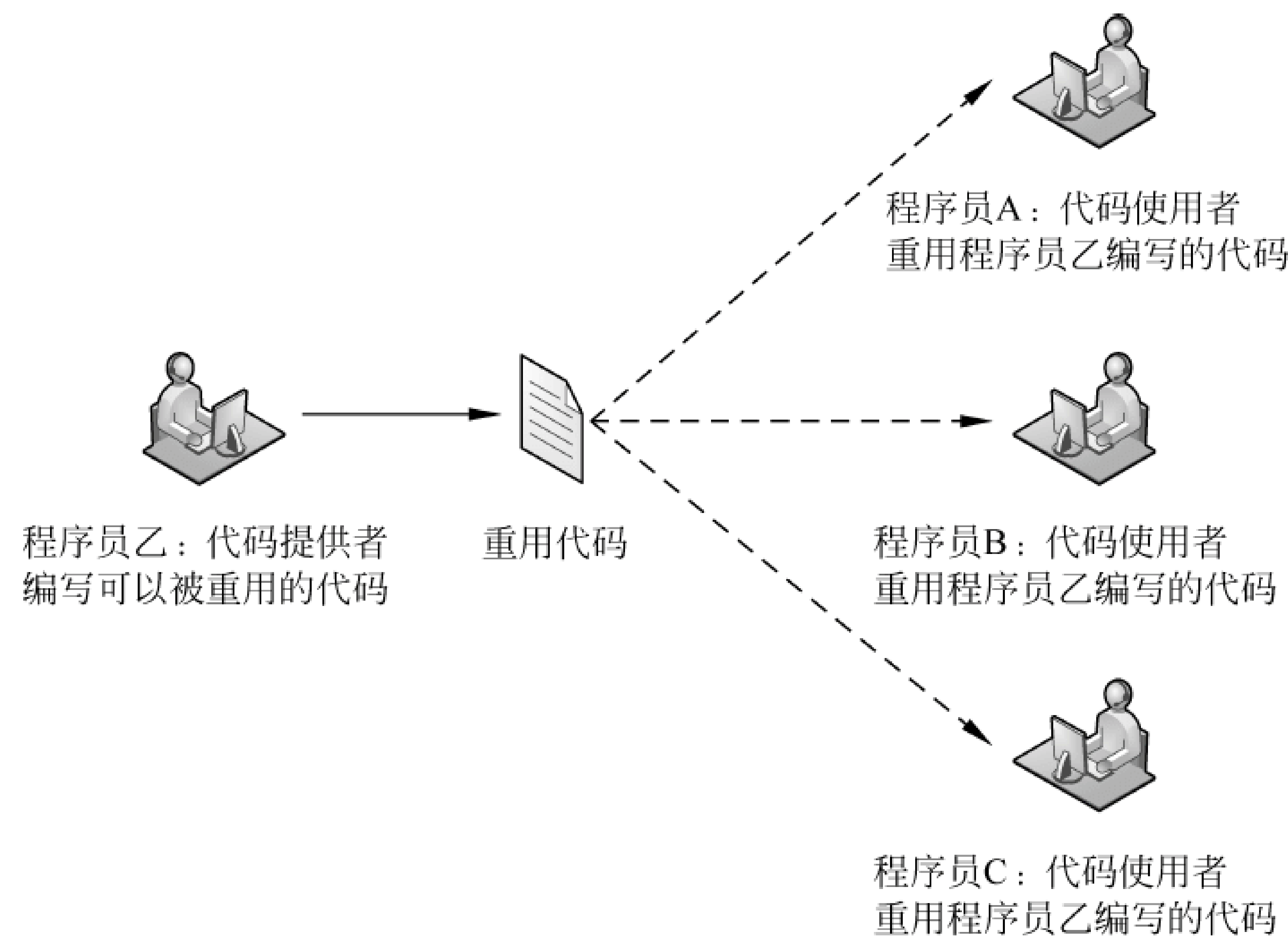


图 3-1 代码重用过程中的两类程序员角色

3.1.2 结构化程序设计中的结构体类型

结构体类型将多个具有内在关联关系的变量组合在一起形成一个逻辑上的整体，变量成为整体的下属成员。定义好的结构体类型将被当作一种新的数据类型来定义变量，所定义出的变量称为结构体变量。

程序员乙通过定义长方形结构体类型可以帮程序员甲再分担一项程序功能，即定义保存数据的变量。修改例 3-1，在程序中引入结构体类型。例 3-2 给出修改后的计算长方形面积和周长的 C++程序代码。

例 3-2 计算长方形面积和周长的 C++程序代码(**结构体类型**)

程序员甲：主函数(1.cpp)

```
 1  #include < iostream >
 2  using namespace std;
 3  #include "2.h"   //插入头文件 2.h
 4
 5  int main()
 6  {
 7      //int a, b; //删除该定义变量语句
 8      //改用结构体类型 Rectangle 定义变量
 9      struct Rectangle  rect;  //定义结构体变量
10
11      cin >> rect.a >> rect.b;
12      //用 rect 的下属成员 a 保存长度
13      //用 rect 的下属成员 b 保存宽度
14
15      //调用子函数求长方形的面积和周长
16      cout << Area( rect.a, rect.b ) << endl;
17      cout << Len( rect.a, rect.b ) << endl;
18      return 0;
19  }
```

程序员乙：子函数(2.cpp)

```
//计算长方形面积和周长的函数
int Area(int length, int width)
{  return ( length * width );  }
int Len(int length, int width)
{  return ( 2 * (length + width) );  }
```

程序员乙：头文件(2.h)

```
//定义一个长方形结构体类型
struct Rectangle
{
    int a;    //保存长度的成员 a
    int b;    //保存宽度的成员 b
};
//声明外部函数的原型
int Area(int length, int width);
int Len(int length, int width);
```

例 3-2 程序的代码说明如下。

(1) 程序员乙定义结构体类型 Rectangle。

变量 a、b 分别保存长方形的长度和宽度。它们都属于长方形数据的一部分，本来就是一个逻辑上的整体。程序员乙将这两个具有内在关联关系的变量组合在一起，在头文件 2.h 中定义一个长方形结构体类型 Rectangle，变量 a、b 成为其下属成员。

和基本数据类型相比，结构体类型是一种由程序员定义出的复杂数据类型，其中可以包含多个变量成员。定义结构体类型就是声明其中包含了哪些变量成员，以及这些变量成员的数据类型。

(2) 程序员甲使用结构体类型 Rectangle 定义变量。

定义好的结构体类型将被当作一种新的数据类型来定义变量，所定义出的变量称为结构体变量。程序员甲使用结构体类型 Rectangle，所定义出的变量 rect 就是一个结构体变量。结构体变量 rect 是一个复杂变量，其中包含两个下属成员，即长度 a 和宽度 b。程序员甲使用这两个下属成员 rect.a 和 rect.b 来分别保存长方形的长度和宽度。

(3) 理解结构体类型。

定义结构体类型的代码描述了一种更高层次上的数据类型。和 int、double 等基本数据类型相比，结构体类型是一种由程序员定义的高级数据类型。

在结构化程序设计中，函数所描述的是一种算法代码。调用函数就是重用算法代码，实现其规定的算法功能。结构体类型的定义代码中包含的是一组定义变量语句，这是一种数据代码。使用结构体类型定义变量，就是重用数据代码，实现其规定的数据管理功能。**结构体类型**是结构化程序设计中重用数据代码的基本语法形式。

结构体类型将程序中大量的数据元素按其内在关联关系划分成一个个相对独立的整体再进行管理，这体现了一种朴素的“**分类管理**”思想。分类可以更好地管理程序代码。

3.1.3 面向对象程序设计中的分类

面向对象程序设计正是基于“分类管理”的思想，在结构化程序设计基础之上所做的进一步发展。面向对象程序设计方法将程序中的数据元素和算法元素按其内在关联关系统一进行分类管理，这就形成了“**类**”(class)。类是结构体类型的进一步扩展，它既可以包含变量，又可以包含函数。类是整体，变量、函数是类的下属成员，分别称为**数据成员**和**函数成员**。

同结构体类型一样，定义好的类将被当作一种新的数据类型来定义变量，用类所定义的变量改称为“**对象**”(object)。

修改例 3-2，在程序中引入类的概念。程序员乙在长方形结构体类型的基础上，进一步将与长方形相关的两个函数 Area()和 Len()也包含进来，使用关键字 class 定义一个长方形类 Rectangle。例 3-3 给出修改后的计算长方形面积和周长的 C++程序代码。

例 3-3 计算长方形面积和周长的 C++程序代码(**类与对象**)

程序员甲：主函数(1.cpp)

```
1  #include <iostream>
2  using namespace std;
3  #include "2.h"   //插入头文件 2.h
4
```

程序员乙：类实现程序文件(2.cpp)

```
#include "2.h"   //插入头文件 2.h
//定义长方形类 Rectangle: 类实现部分
//给出各函数成员的完整定义代码
int Rectangle:: Area() //不需要传递长宽
```

```
 5  int main()
 6  {
 7      //struct Rectangle  rect;  //删除该语句
 8      //改用类 Rectangle 定义变量(即对象)
 9       Rectangle  rect;//定义一个长方形对象 rect
10
11      cin >> rect.a >> rect.b;
12      //用 rect 的数据成员 a 保存长度
13      //用 rect 的数据成员 b 保存宽度
14
15      //调用 rect 的函数成员求其面积和周长
16      //调用时不需要传递长宽数据
17      cout << rect.Area() << endl;
18      cout << rect.Len() << endl;
19      return 0;
20  }
```

```
{   return ( a * b );  }
int Rectangle:: Len()
{   return ( 2 * (a + b) );   }
程序员乙:类声明头文件(2.h)
//定义一个长方形类 Rectangle
//定义类的代码分为声明和实现两部分
class Rectangle   //类声明部分
{
public:
    int a;        //数据成员:保存长度
    int b;        //数据成员:保存宽度
    int Area();   //函数成员:计算面积
    int Len();    //函数成员:计算周长
};
//类 Rectangle 的实现部分放在 2.cpp 文件中
```

例 3-3 程序的代码说明如下。

(1) 程序员乙定义长方形类 Rectangle。

程序员乙将与长方形相关的两个变量(a、b)和两个函数(Area、Len)组合在一起,定义一个长方形类 Rectangle。长方形类 Rectangle 是一个整体,变量 a、b 是其数据成员,函数 Area()和 Len()是其函数成员。

定义类的代码分为两部分,分别是**类声明**部分和**类实现**部分。程序员乙将 Rectangle 类声明部分的代码保存在**头文件** 2.h 中,将类实现部分的代码保存在**程序文件** 2.cpp 中。

- **类声明**(class declaration)。使用关键字 **class** 并指定类名,并在随后的大括号中声明所包含的数据成员和函数成员。声明函数成员就是声明其原型,完整的函数定义代码被放在类实现部分。

  注:类声明部分有一个关键字 public,其语法作用将在 3.1.4 节中再做讲解。
- **类实现**(class implementation)。在类实现部分给出各函数成员的完整定义代码。定义时需在函数名前加类名"Rectangle::"进行限定,指明该函数是属于 Rectangle 类的。其中的"::"为两个冒号,被称为**作用域运算符**(或作用域分辨符)。

在类定义中,数据成员(例如变量 a、b)相当于是类中的全局变量,函数成员可以直接访问它们。例如,函数 Area()和 Len()在计算长方形面积和周长时直接从数据成员 a、b 中读取长、宽值,因此这两个函数不再需要定义形参来接收长、宽数据。以类的形式来组织程序代码,可以有效减少函数间的参数传递。

从类的定义代码可以看出,类是一种由程序员定义的高级数据类型(被称为**类类型**)。其中描述了类包含哪些数据成员以及各成员的数据类型(这属于数据代码),同时还以函数成员的语法形式描述了类具有哪些处理算法(这属于算法代码)。

(2) 程序员甲使用长方形类 Rectangle 定义对象。

类是结构体类型与函数的结合体,其中既包含数据代码,又包含算法代码。类是一种由程序员定义的高级数据类型,用类所定义的变量改称为对象。

程序员甲编写主函数时,使用长方形类 Rectangle 定义一个对象 rect。rect 被称为是一

个长方形类的对象。按照长方形类 Rectangle 的定义，对象 rect 将包含两个数据成员（即 rect.a 和 rect.b)，程序员甲使用这两个数据成员来分别保存长方形 rect 的长度和宽度。对象 rect 还包含两个函数成员，即 rect.Area()和 rect.Len()，程序员甲调用这两个函数成员来分别计算长方形对象 rect 的面积和周长。

使用类定义对象，然后访问对象的成员，这实际上是重用该类的代码，实现其规定的程序功能。程序员乙编写好的长方形类 Rectangle 可以在本次项目中使用，也可以在今后的项目中使用，或提供给任何其他程序员使用。不管是什么项目，或是哪位程序员，只要使用这个类就都能轻松实现求长方形面积或周长的功能。类代码也是**“一次编写，长期使用”**。重用类代码时，既重用了数据代码，又重用了算法代码。在面向对象程序设计中，**类**是重用“数据代码＋算法代码”的基本语法形式。

针对计算长方形面积和周长的问题，例 3-2、例 3-3 分别采用结构化程序设计方法和面向对象程序设计方法，进而设计出了不同的程序。这两种方法有什么不同之处呢？程序设计方法主要应用于大型软件的设计开发。将大型程序中的数据元素和算法元素分解成程序零件，将不同零件的设计任务交由不同的程序员完成，这样就能以团队的形式来共同开发，然后将开发好的零件组装在一起，最终完成复杂的程序功能。目前，程序设计方法分为结构化程序设计和面向对象程序设计两种，它们分别采用不同的方式来分解和组装程序零件。

1. 结构化程序设计方法

结构化程序设计方法将程序中的复杂算法分解成多个函数，**函数**是分解出的**算法零件**。结构化程序设计方法将大量保存数据的变量按其内在关联关系划分成一个个相对独立的结构体类型，**结构体类型**是分解出的**数据零件**。从例 3-2 可以看出，结构化程序设计方法所分解出的算法零件和数据零件是分离的。这种分离的零件会带来什么后果呢？我们结合这个例子来做进一步分析。

例 3-2 将计算长方形面积和周长的算法分解成两个算法零件（即函数 Area()和 Len()），将保存长方形长、宽数据的变量 a、b 组合在一起形成一个数据零件（即结构体类型 Rectangle）。按这种方式所分解出的算法零件和数据零件在语法上是相互独立的（即相互分离的）。

假设例 3-2 在编写完成后，程序员乙发现长方形结构体类型 Rectangle 中保存长、宽数据的变量 a、b 被定义成了 int 型，只能处理整数。而实际应用中长方形的长、宽数据经常是实数，程序员乙希望对这个数据零件进行升级，将变量 a、b 的数据类型修改为 double 类型。程序员乙按如下形式修改头文件 2.h 中结构体类型 Rectangle 的定义代码：

```
struct Rectangle                              //修改成员 a、b 的数据类型
{
    double a;                                 //原来为 int a;
    double b;                                 //原来为 int b;
};
```

将数据零件中的数据类型改为 double 后，程序员乙发现还需要继续修改算法零件，即修改函数 Area()和 Len()，必须将这两个函数的形参和返回值类型同步都改为 double 型。修改后的函数代码如下：

```
double Area( double length, double width)        //原来为 int Area(int length, int width)
{   return ( length * width );   }
double Len( double length, double width)         //原来为 int Len(int length, int width)
{   return ( 2 * (length + width) );   }
```

可以看出，虽然数据零件和算法零件是相互分离的，但仍然互相耦合，互相影响。因为函数 Area()和 Len()的定义代码改变了，其头文件 2.h 中对应的原型声明代码也要做相应修改。修改后的原型声明代码如下：

```
double Area( double length, double width);       //原来为 int Area(int length, int width);
double Len( double length, double width);        //原来为 int Len(int length, int width);
```

可以看出，如果想要修改结构化程序中的数据（例如修改数据的类型、添加或减少数据项等），程序员除了修改对应的变量定义语句之外，还需要修改所有与之关联的函数定义、声明和调用代码。

大型软件系统包含成千上万个函数，其中还存在多层嵌套调用关系。这些函数是由不同程序员编写的，被分散保存在不同的程序文件中。修改数据所造成的连带修改范围会很广，也可能会涉及很多位程序员，修改难度将非常大。造成上述问题的原因就在于结构化程序设计方法将本来具有关联关系的数据和算法割裂开了。

2. 面向对象程序设计方法

为了解决结构化程序设计方法的不足，面向对象程序设计方法在分解程序零件时，不是单纯分解算法或数据，而是将数据和与之关联的算法划分在一起形成“**数据类**”。数据类是“数据＋算法”，是数据及其处理算法的完整描述。

数据类将具有关联关系的变量和函数集中定义在一起，不管是数据或算法，修改时通常只要修改该数据类的代码即可，与其他代码无关。面向对象程序设计方法就是按照数据类来分解和编写程序的。

例如，例 3-3 中的长方形类 Rectangle 就是按上述方法所分解出的一个数据类。在这个例子中，如果程序员乙希望对程序进行升级，将长方形类 Rectangle 中变量 a、b 的数据类型修改为 double 类型，则只要修改长方形类 Rectangle 中相关的代码。修改后长方形类 Rectangle 的定义代码如下：

```
class Rectangle                                  //修改头文件 2.h 中的类声明部分
{
public:
    double a;                                    //原来为 int a;
    double b;                                    //原来为 int b;
    double Area();                               //原来为 int Area();
    double Len();                                //原来为 int Len();
};
//修改程序文件 2.cpp 中的类实现部分
double Rectangle::Area()                         //原来为 int Rectangle::Area()
{   return ( a * b );   }
double Rectangle::Len()                          //原来为 int Rectangle::Len()
{   return ( 2 * (a + b) );   }
```

例 3-3 以类的形式来组织程序代码，其中的函数成员 Area()和 Len()都没有形参，修改时只需修改返回值类型即可，因此代码修改量减少了。更为重要的是，在修改类中数据成员时，其连带修改的代码一般不会超出本类的范围。一个类通常是由一位或少数几位程序员编写的，修改所牵涉的程序员比较少。因此以类的形式来组织程序代码，今后的修改难度将大大降低。

3.1.4 面向对象程序设计中的封装

从例 3-1 演变到例 3-3，程序员乙已经承担了 4 项程序功能中的 2 项，即定义保存数据的变量和数据处理。程序员乙是编写重用代码的程序员，我们希望他的重用代码能完成更多的程序功能。下面我们继续挖掘程序员乙的潜能，让他帮助程序员甲完成剩余的 2 项程序功能，即输入原始数据和输出处理结果。

程序员乙继续在例 3-3 的长方形类 Rectangle 中添加函数成员 Input()和 Output()，实现输入原始数据和输出处理结果的功能。例 3-4 给出添加函数成员后的计算长方形面积和周长的 C++程序代码。

例 3-4 计算长方形面积和周长的 C++程序代码(**添加输入输出功能**)

程序员甲：主函数(1.cpp)

```
1   //#include < iostream > //删除这 2 条语句
2   //using namespace std;
3   //主函数不再使用 cin/cout,删除上面 2 条语句
4
5   #include "2.h"   //插入头文件 2.h
6
7   int main()
8   {
9       //使用功能完善后的类 Rectangle 定义对象
10      Rectangle rect;//定义一个长方形对象 rect
11
12      //cin >> rect.a >> rect.b;//删除该语句
13      rect.Input();//调用 Input 成员输入长、宽
14
15      //删除下面两条语句
16      //cout << rect.Area() << endl;
17      //cout << rect.Len() << endl;
18      rect.Output(); //调用 Output 成员输出结果
19
20      return 0;
21  }
22
23
24
25
26
27
```

程序员乙：类实现程序文件(2.cpp)

```
#include < iostream >
using namespace std;
//本程序需使用 cin/cout,添加上面两条语句
#include "2.h"   //插入头文件 2.h
//定义长方形类 Rectangle:类实现部分
double Rectangle::Area()
{   return ( a * b );   }
double Rectangle::Len()
{   return ( 2 * (a + b) );   }
void Rectangle:: Input()   //输入长、宽
{   cin >> a >> b;   }
void Rectangle:: Output()//输出面积和周长
{
    cout << Area() << endl;
    cout << Len() << endl;
}
```

程序员乙：类声明头文件(2.h)

```
//为长方形类 Rectangle 添加两个函数成员
class Rectangle      //类声明部分
{
public:
    double a, b;     //数据成员:保存长、宽
    double Area();   //函数成员:计算面积
    double Len();    //函数成员:计算周长
    void Input();    //函数成员:输入长、宽
    void Output();   //函数成员:输出结果
};
```

例 3-4 程序的代码说明如下。

(1) 程序员乙继续完善长方形类 Rectangle 的功能。

程序员乙在例 3-3 长方形类 Rectangle 的基础上，通过添加函数成员 Input()实现了输入原始数据(即长方形的长和宽)的功能，再添加函数成员 Output()又实现了输出处理结果(即显示长方形的面积和周长)的功能。

(2) 程序员甲使用功能完善后新的长方形类 Rectangle。

程序员甲编写主函数的目的是为了输入长方形的长、宽，然后计算其面积和周长。使用功能完善后新的长方形类 Rectangle，程序员甲在主函数中仅仅编写如下 3 条语句就实现了所需要的程序功能。

```
Rectangle  rect;              //定义一个长方形对象 rect
rect.Input();                 //调用长方形对象 rect 的函数成员 Input()输入其长和宽
rect.Output();                //调用长方形对象 rect 的函数成员 Output()显示其面积和周长
```

其中，调用函数成员 Input()的目的是输入长方形对象 rect 的长和宽。为什么调用函数成员 Input()就能输入长和宽呢？因为程序员乙在定义 Input()函数成员时编写了如下 cin 语句：

```
cin >> a >> b;                //输入长方形的长和宽
```

同理，调用函数成员 Output()就能输出长方形对象 rect 的面积和周长，因为程序员乙在定义 Output()函数成员时编写了如下 cout 语句：

```
cout << Area() << endl;       //调用函数成员 Area()计算面积，并通过 cout 显示出来
cout << Len() << endl;        //调用函数成员 Len()计算周长，并通过 cout 显示出来
```

至此，程序员乙所编写的类 Rectangle 已完成了处理长方形数据所需的全部 4 项功能。可以看出，随着所承担功能的增多，程序员乙编写的代码越来越长，反过来程序员甲编写的代码则越来越短。程序员乙编写的长方形类 Rectangle 是能被重用的代码，其功能越强，重用的价值就越大。

仔细分析例 3-4 的程序，程序员乙编写的长方形类 Rectangle 总共包含了 6 个成员，分别是 2 个数据成员 a、b，以及 4 个函数成员 Area()、Len()、Input()和 Output()。但程序员甲在使用 Rectangle 类时，只需要用 Input()和 Output()这两个成员就能完成所需要的程序功能，即输入长、宽，然后输出面积、周长。换句话说，程序员甲在使用 Rectangle 类时不需要直接访问另外 4 个成员，即数据成员 a、b，函数成员 Area()、Len()。

注意：不需要直接访问的成员并不意味着是无用的成员，因为它们会被间接访问。例如，程序员甲在调用函数成员 Input()时会间接访问数据成员 a、b，在调用函数成员 Output()时会间接调用函数成员 Area()和 Len()。

定义类的程序员可以将需要被外部直接访问的成员开放出来，同时将不需要被直接访问的成员隐藏起来，这就是面向对象程序设计中类的**封装**(encapsulation)。

1. 类的封装

基于分类管理的思想，面向对象程序设计方法将程序中具有内在关联关系的变量和函数组合在一起形成类。类是一个整体，变量和函数是类的下属成员。类的封装有两层含义。

- **开放**。定义类时将必须被外部访问的成员开放出来,以保证类的功能可以被正常使用。
- **隐藏**。定义类时将不需要被外部访问的成员隐藏起来,以防止它们被误访问。被隐藏的成员只能在内部使用,被类里的其他成员访问,在类的外部不能直接访问。

封装是一种常见的防护方法。例如设计电视机时要考虑电视机内部有很多电子元器件,需要用一个外壳将它们封装起来,这样可以防止用户误操作。同时将使用电视机所必须的操作(例如切换频道、调节音量等)通过一些按键开放出来,这样用户就可以使用这些按键来操作电视机。

面向对象程序设计中,封装是通过为类成员赋予不同的**访问权限**来实现的。访问权限主要有两种,它们分别是:

- **公有权限**(public)。被赋予公有权限的类成员是开放的,称为公有成员。
- **私有权限**(private)。被赋予私有权限的类成员将被隐藏,称为私有成员。

编写类的程序员通过设定访问权限将类成员封装起来,将需要访问的成员开放出来,不需访问的成员隐藏起来,这样可以避免误访问。访问权限是编写类的程序员为保证其他程序员正确访问类成员所采取的一种防护措施。

公有成员是封装后类提供给外界的操作接口。通常一个类必须有公有成员,否则这个类无法使用。程序员设计类时应根据功能要求合理设定成员权限,一方面要开放用户正常使用所必需的成员,另一方面要尽可能隐藏不需要被直接访问的成员。

2. 封装长方形类 Rectangle

程序员乙封装长方形类 Rectangle,就是在其类声明部分为各成员设定访问权限。在例 3-4 中,程序员乙将长方形类 Rectangle 的所有成员都设定为公有权限 public,即将它们都开放出来,这相当于没有封装。

下面程序员乙将根据例 3-4 中长方形类 Rectangle 的功能要求,重新设定各成员的访问权限。将需要被外部访问的 2 个成员 Input()和 Output()设为公有权限 public,即将它们开放出来;将不需要被外部直接访问的另外 4 个成员 a、b、Area()和 Len()设定为私有权限 private,即将它们隐藏起来。图 3-2 分别给出长方形类 Rectangle 修改前后的封装示意图,其中类成员名前面的"+"表示公有权限(即对外开放的接口),"-"表示私有权限(被隐藏了)。

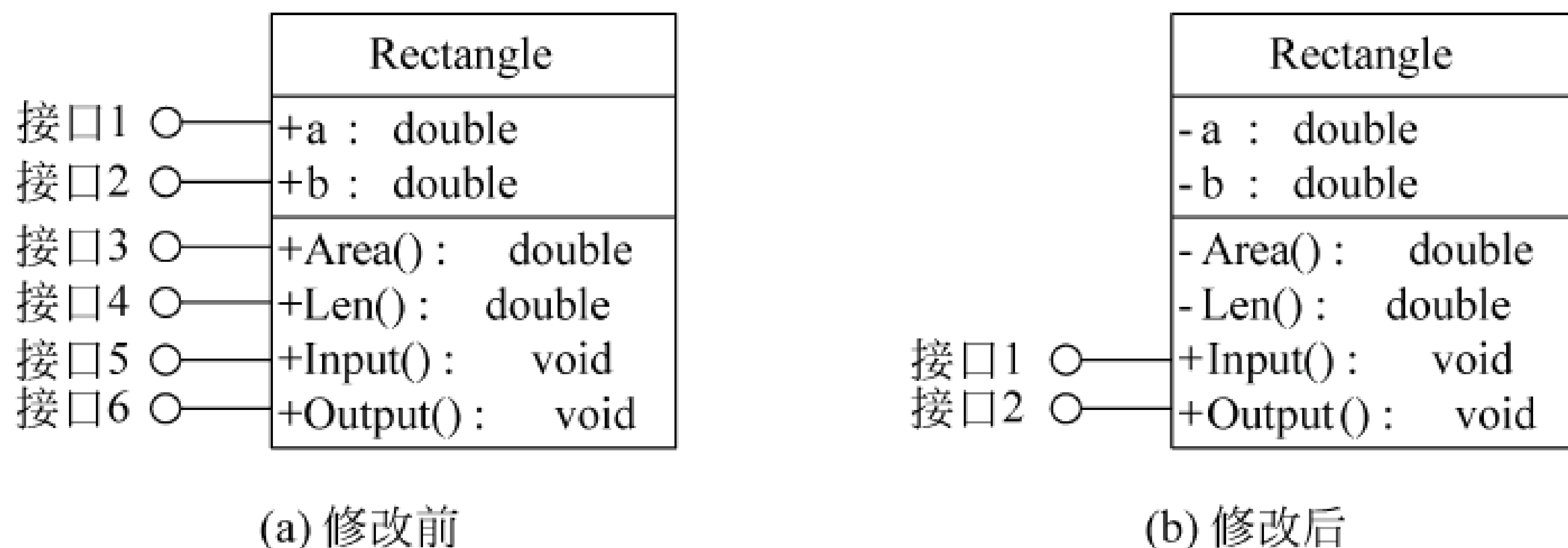


图 3-2 例 3-4 中长方形类 Rectangle 修改前后的封装示意图

重新设定访问权限后,例 3-4 中长方形类 Rectangle 声明部分的代码被修改成如下形式:

```
class Rectangle                    //在类声明部分设定各成员的访问权限
{
private:                           //以下 4 个成员被设定为私有权限
    int a, b;                      //数据成员:保存长和宽
    int Area();                    //函数成员:计算面积
    int Len();                     //函数成员:计算周长
public:                            //以下 2 个成员被设定为公有权限
    void Input();                  //函数成员:输入长和宽
    void Output();                 //函数成员:输出面积和周长
};
```

C++语言中的访问权限只在类声明部分设定,与类实现部分的代码无关。重新设定长方形类 Rectangle 的访问权限,不需要修改其类实现部分的代码。

3. 封装的作用

在程序员乙重新设定访问权限之后,程序员甲使用新的长方形类 Rectangle 定义对象,将只能访问对象中的公有成员。例如:

```
Rectangle  rect;                   //定义一个长方形对象 rect
rect.Input();                      //调用公有的函数成员 Input(),输入长方形的长和宽
rect.Output();                     //调用公有的函数成员 Output(),输出长方形的面积和周长
```

可以看出,程序员甲使用新的长方形类 Rectangle 仍能够实现正常的程序功能。这说明程序员乙所设定的访问权限是合理的,他将程序员甲正常使用所必需的成员都开放出来了。

另外,在程序员乙重新设定访问权限之后,程序员甲将不能访问长方形类对象 rect 中的私有成员。例如,程序员甲需要从键盘输入长方形对象 rect 的长和宽,这时他只能调用公有函数成员 Input()来实现,而不能用 cin 指令直接输入。例如,下面这条输入语句在编译时会提示语法错误,因为不能访问对象 rect 的私有成员 a、b。

```
cin >> rect.a >> rect.b;           //访问私有成员,编译时编译器将提示语法错误
```

可以看出,虽然长方形对象 rect 包含私有成员 a、b,但是不能访问。程序员乙将数据成员 a、b 设为私有成员的目的是:迫使程序员甲只有通过公有的函数成员 Input()才能输入长、宽数据。这么做有什么好处呢?请看下面经过改进的 Input()函数:

```
void Rectangle:: Input()           //输入长方形的长和宽
{
    cout << "请输入长和宽:";  cin >> a >> b;
    while ( a < 0 || b < 0 )       //数据合法性检查
    {  cout <<"长、宽值不能为负数,请重新输入:";  cin >> a >> b;  }
}
```

这个新的 Input()函数对所输入的长、宽数据进行合法性检查,要求它们的数值不能是负数。强制通过这个 Input()函数来输入长、宽数据,可以保证所输入的数值都是合法有效的。

3.1.5 Java 语言中的类与对象

采用结构化程序设计方法(如例 3-1、例 3-2),可以采用 C 语言或 C++语言编写程序,因为它们都支持结构化程序设计方法。如果采用面向对象程序设计方法(如例 3-3、例 3-4),则必须改用支持面向对象的程序设计语言,例如 C++或 Java。下面采用面向对象程序设计方法,改用 Java 语言来编写前面的计算长方形面积和周长的程序(见例 3-5)。

例 3-5 计算长方形面积和周长的 Java 程序代码(**类与对象**)

程序员甲:主类文件(RectangleTest.java)

```
1  public class RectangleTest {   //主类
2
3    //将主函数 main()定义在类中
4    public static void main(String[] args) {
5      //Java 需要动态创建对象
6      Rectangle obj = new Rectangle();
7
8      obj.Input();        //输入长、宽
9      obj.Output();       //显示结果
10   }
11
12 }
```

程序员乙:长方形类文件(Rectangle.java)

```
1  import java.util.Scanner;//导入外部程序 Scanner
2
3  public class Rectangle {  //长方形类定义代码
4    private double a, b;    //字段:保存长和宽
5    private double Area()   //方法:计算面积
6    {  return a * b;  }
7    private double Len()    //方法:计算周长
8    {  return 2 * (a + b);  }
9
10   public void Input() {   //方法:输入长、宽
11     //创建键盘扫描器对象
12     Scanner sc = new Scanner( System.in );
13     //然后通过键盘扫描器对象输入长、宽
14     a = sc.nextDouble();  b = sc.nextDouble();
15   }
16   public void Output() {  //方法:输出结果
17     System.out.println( Area() + ", " + Len() );
18   }
19 }
```

例 3-5 程序的代码说明如下。

(1) 主函数 main()。

主函数 main()是程序执行的起点。一个可以执行的程序必须包含主函数。Java 程序主函数的定义格式为:

```
public static void main(String[]args) {
    …  //此处定义主函数的代码
}
```

Java 语言是“纯”面向对象的程序设计语言,程序中没有游离在类外的全局变量或函数。Java 程序中所有的变量和函数都必须被定义在某个类中,包括主函数 main()。

(2) 字段和方法。

Java 语言将类中定义的**数据成员**改称为“**字段**”,函数成员改称为“**方法**”。本书后续章节将基本遵照这两个 Java 语言的称呼,例如将“主函数”改称为“**主方法**”。某些场合为便于解释语法,也会使用“数据成员”“函数成员”或“函数”这样的说法。

(3) 主类。

包含主方法 main()的类被称为“**主类**”。例 3-5 将主方法定义在了类 RectangleTest

中，这个类就是主类。在例 3-5 中，主类 RectangleTest 只定义了一个主方法，其目的是为了测试或使用长方形类 Rectangle 的功能。

初学 Java 编程，经常需要单独编写一个主类来测试自己所编写的 Java 类，因此在教学时也将这样的主类称为**“测试类”**。

(4) 文件名与类名。

程序员通常将一个 Java 类单独保存到一个源程序文件中。这时，源程序的文件名应当与类名一致。例如，例 3-5 中保存主类 RectangleTest 的文件名为 RectangleTest.java，保存长方形类 Rectangle 的文件名为 Rectangle.java。

Eclipse 集成开发环境在新建 Java 类时，会自动创建一个与类名相同的源程序文件(扩展名为.java)，用于保存这个类的定义代码。

(5) 运行 Java 程序。

运行 Java 程序，就是执行主类中的主方法 main()。主方法 main()是程序执行的起点。在 Eclipse 集成开发环境中运行 Java 程序，选择菜单 **Run** 下的子菜单 **Run**。

(6) 将主方法 main()直接定义到自己的 Java 类中。

可以将例 3-5 中的主方法直接定义到长方形类 Rectangle 中，这样可以简化程序代码(例 3-6)。

注：建议初学者先单独编写主类，将主方法与自己的 Java 类分开，这样的代码结构比较清楚，易于理解。

例 3-6　将主方法 main()定义在长方形类 Rectangle 中

```
1   import java.util.Scanner;                          //导入外部程序 Scanner
2
3   public class Rectangle {                           //长方形类定义代码
4       private double a, b;                           //字段:保存长和宽
5       private double Area()  {   …   }               //代码省略
6       private double Len()   {   …   }               //代码省略
7       public void Input()    {   …   }               //代码省略
8       public void Output()   {   …   }               //代码省略
9
10      //将主方法 main()定义在长方形 Rectangle 类中
11      public static void main(String[] args) {
12          Rectangle obj = new Rectangle();
13          obj.Input();                               //输入长、宽
14          obj.Output();                              //显示结果
15      }
16      //语法上,主方法也是类的一个成员,放在其他类成员的前面或后面都可以
17  }
```

本节通过具体的程序实例介绍了结构化程序设计到面向对象程序设计的演变过程，相信读者对这两种程序设计方法已经有了比较直观的认识。

自 1965 年提出结构化程序设计概念，再到 1989 年 ANSI 发布第一个 C 语言标准(ANSI C 或 C 89)，结构化程序设计方法风靡全球。C 语言支持结构化程序设计方法，C 语言曾在很长一段时间内是全球使用最为广泛的计算机语言。

但到了 1998 年，面向对象程序设计的 C++语言被 ISO 和 ANSI 两大标准化组织同时批准为国际标准(ISO/IEC 14882 或 C++98)，由此程序设计便迈入了面向对象的时代。

1995 年,Java 语言正式推出。随着互联网的普及,Java 语言在网络应用程序开发方面取得了巨大成功,这就从根本上确立了面向对象程序设计的主导地位。

面向对象程序设计在结构化程序设计的基础上,引入了**分类**(抽象)、**封装**、**继承**和**多态**的思想,将程序设计方法推向了一个新的高度。面向对象程序设计方法可以更好地组织和管理程序代码,也更便于重用代码。本章先介绍分类和封装,下一章再介绍继承和多态。

本节习题

1. 下列关于类的描述中,错误的是(　　)。
   A. 类可认为是一种高级数据类型　　B. 用类所定义出的变量称为对象
   C. 类包含数据成员和函数成员　　D. 类是结构化程序设计中的概念
2. 下列关于重用代码的描述中,错误的是(　　)。
   A. 函数是重用算法代码的语法形式
   B. 结构体类型是重用数据代码的语法形式
   C. 类是同时重用算法代码和数据代码的语法形式
   D. 类是一种数据类型,因此只能重用数据代码
3. 关于程序开发过程中的程序员角色,下列描述中错误的是(　　)。
   A. 一个程序员可以为其他程序员提供代码,即代码提供者
   B. 一个程序员可以使用其他程序员提供的代码,即代码使用者
   C. 一个程序员可以既是代码提供者,又是代码使用者
   D. 一个程序员不能既是代码提供者,又是代码使用者
4. 关于程序设计方法,下列描述中错误的是(　　)。
   A. 程序设计方法是研究如何对大型程序设计任务进行分解的方法
   B. 结构化程序设计分解出的函数是一种算法零件
   C. 结构化程序设计分解出的结构体类型是一种数据零件
   D. 面向对象程序设计分解出的类是一种数据零件
5. 下列选项中,(　　)不属于面向对象程序设计的核心思想。
   A. 抽象　　B. 封装　　C. 继承　　D. 模块化

3.2 面向对象程序的设计过程

程序员拿到一个具体的设计任务,该如何运用面向对象的方法进行程序设计呢?本节以一个测算养鱼池工程总造价的设计任务为例,具体讲解面向对象程序设计方法的设计过程。程序设计任务如下:公园计划修建一个长方形观赏鱼池,另外配套修建一大一小两个圆形蓄水池,分别存放清水和污水(见图 3-3)。养鱼池和蓄水池的造价均为每平方米 10 元。请设计一个测算养鱼池工程总造价的计算机程序。

面向对象程序的设计过程,简单地可分为**分析**、**抽象**和**组装** 3 个阶段。本节以**统一建模语言**(Unified Modeling Language,UML)来描述设计结果,然后再用 Java 语言将设计结果编写成计算机程序。



图 3-3　测算养鱼池的工程总造价

3.2.1　分析

面向对象程序设计的第一阶段是需求分析。程序员经过需求调研可以知道，使用计算机测算养鱼池工程总造价的工作流程大致如下。

(1) **用户**启动**测算程序**，由计算机执行这个程序。

(2) 程序等待用户输入原始数据，其中包括养鱼池的长和宽、清水池和污水池的半径。用户输入数据后，程序继续执行。

(3) 程序要在计算机中模拟创建**养鱼池**、**清水池**和**污水池**，然后计算并汇总其造价。

(4) 显示汇总后的工程总造价。测算程序结束。

采用面向对象程序设计方法，程序员通常以**用例图**的形式来描述工作流程和功能需求，然后对工作流程进行分析，从中提取出所有参与流程的**客观事物**，并以顺序图、活动图和状态图等形式描述各客观事物是如何参与工作流程的。例如，从测算养鱼池工程总造价的工作流程中共提取出用户、测算程序、养鱼池、清水池和污水池等 5 个客观事物。使用顺序图来描述它们如何参与测算养鱼池工程总造价的流程，如图 3-4 所示。

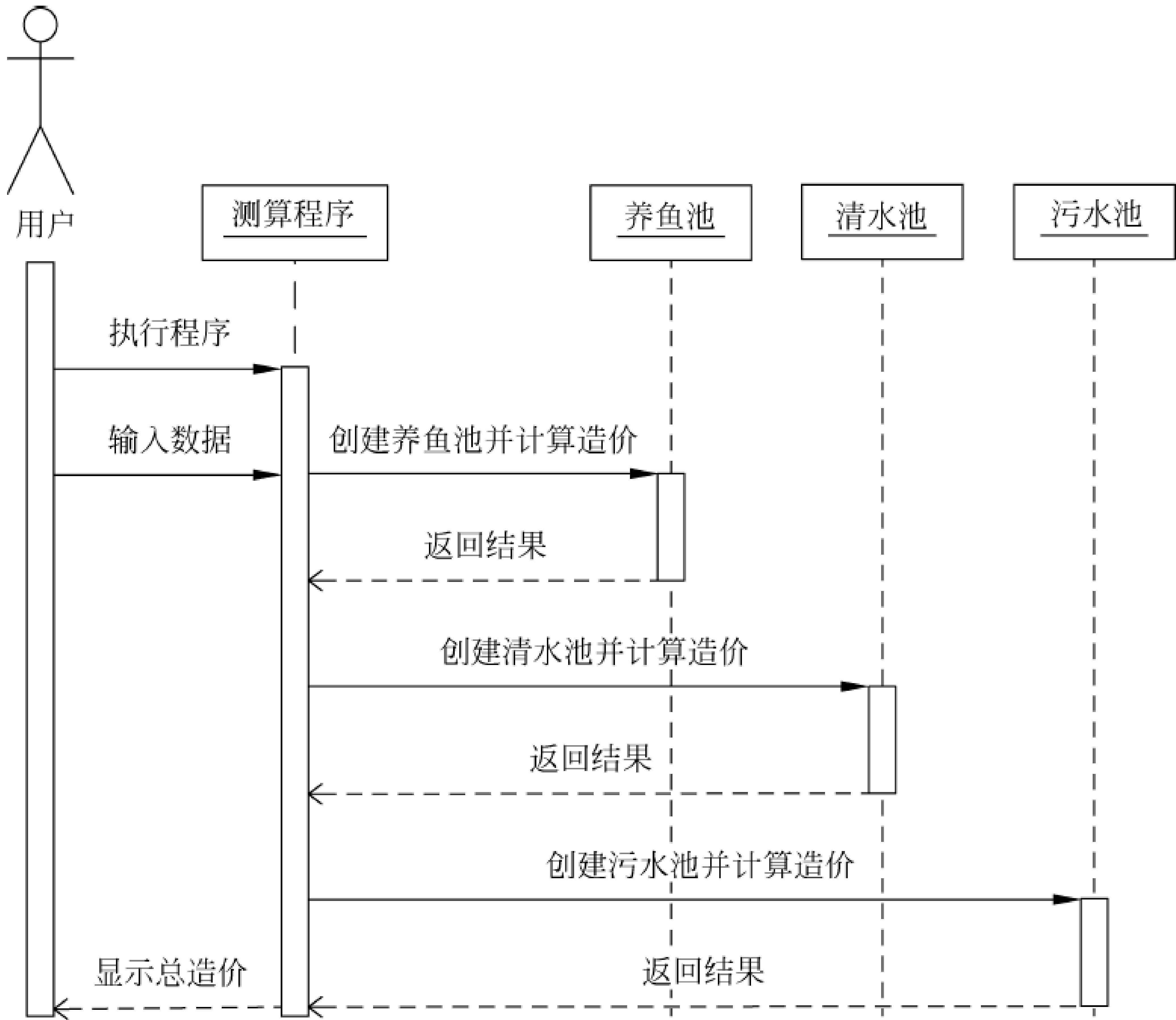


图 3-4　测算养鱼池工程总造价顺序图示例

图 3-4 中的客观事物“测算程序”是指一个程序设计任务，即测算养鱼池工程总造价的程序。在 Java 语言中，这意味着要编写一个描述测算养鱼池工程总造价流程的主方法 main()。主方法是与程序设计任务相关的，每个程序设计任务都要编写自己的主方法。

在测算养鱼池工程总造价程序中，还要对养鱼池、清水池和污水池这 3 个客观事物进行处理，求出它们的造价。下面问题的关键是如何在程序中描述养鱼池、清水池和污水池这 3 个客观事物，以及如何处理它们？

使用结构化程序设计方法，可以将客观事物分解成数据和算法两部分。计算机程序使用变量来保存描述客观事物的数据，使用函数来描述处理客观事物的算法。例如在程序中描述和处理长方形养鱼池，程序员可以编写如下代码：

```
double length, width;                    //定义 2 个变量，分别保存长方形养鱼池的长、宽数据
double RectCost(double a, double b)      //定义 1 个函数，描述计算养鱼池造价的算法
{
    double cost ;
    cost = a * b * 10 ;                  //造价 = 长 × 宽 × 单价
    return cost ;
}
```

可以看出，计算机程序要处理客观世界中的事物，首先要对客观事物进行**抽象**，提炼出数据模型。程序设计对客观事物的处理，实际上是对其数据模型的处理。3.2.2 节将介绍面向对象程序设计是如何将客观事物抽象成数据模型的。

3.2.2 抽象

面向对象程序设计的第二阶段是对客观事物进行抽象。计算机只能进行数值计算，要想让计算机来处理客观世界中的事物，首先需要为客观事物建立数据模型。面向对象程序设计中的数据模型应包含以下两方面的内容。

(1) 属性(property)。

为描述清楚客观事物，需要哪些**数据**？描述事物的数据被称为**属性**或**状态**(state)。例如在测算养鱼池工程总造价程序中，描述长方形养鱼池需要长和宽这两个属性。计算机程序通过定义**变量**来存储属性数据。变量就是数据模型中的属性。

(2) 方法(method)。

对客观事物要进行什么样的**处理**？描述事物处理的算法被称为**方法**。例如在测算养鱼池工程总造价程序中，对长方形养鱼池的处理就是计算其造价。计算机程序通过定义**函数**来描述算法。**函数**就是数据模型中的方法。

使用面向对象程序设计方法为长方形养鱼池建立数据模型，其中应当包含两个属性(即长和宽)，以及一个计算造价的方法。

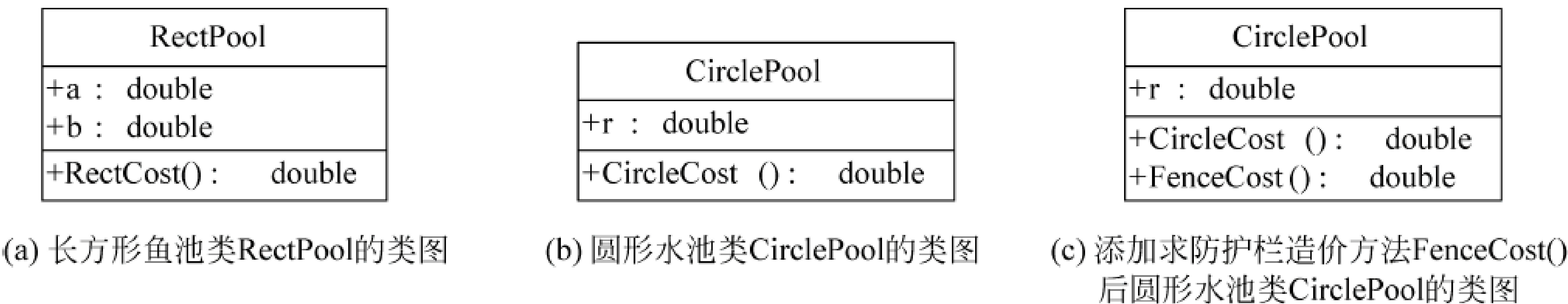
在测算养鱼池工程总造价程序中，还需要处理圆形清水池和圆形污水池。凭直觉，这两个水池有点类似。是的，圆形清水池和圆形污水池具有相同的数据模型，它们都包含一个属性“半径”和一个“求圆形水池造价”的方法。

面向程序设计将客观世界中的事物称为一个个具体的**客观对象**。将具有相同数据模型的客观对象归纳成一类，这就是面向对象程序设计中的**分类**。对客观事物进行归纳，划分成不同的类，这是人类认识客观世界、解决实际问题常用的思维方法。分类就是抓住主要特

征，忽略次要特征，将具有共性的事物划分成一类。分类的过程是一个不断抽象的过程，面向对象程序设计将“分类”称为**抽象**。

面向对象程序设计将抽象得到的数据模型称为**类**，其中包含属性成员和方法成员。每个类都代表了一组客观世界中具有共性的事物，它是这组客观事物抽象后得到的数据模型。进一步完善数据模型，合理设定各成员的访问权限，这就是类的**封装**。面向对象程序设计方法用**类图**来描述类的设计结果。

在测算养鱼池工程总造价程序中，长方形养鱼池被抽象成一个长方形鱼池类(假设命名为RectPool)，圆形清水池和圆形污水池被抽象成一个圆形水池类(假设命名为CirclePool)。用类图描述上述设计结果，图3-5(a)是长方形鱼池类RectPool的类图，图3-5(b)是圆形水池类CirclePool的类图。



(a) 长方形鱼池类RectPool的类图　(b) 圆形水池类CirclePool的类图　(c) 添加求防护栏造价方法FenceCost()后圆形水池类CirclePool的类图

图3-5　类图示例

一个类到底要提炼多少属性和方法，这要依据程序的功能要求而定。假设公园修建的清水池和污水池还要增加防护栏，那么就需要为圆形水池类增加一个求防护栏造价的方法。图3-5(c)给出添加求防护栏造价方法FenceCost()后圆形水池类CirclePool的类图。还可以继续为类添加功能，对类进行合理封装，优化类的设计，这样可以方便今后类的使用。

面向对象程序的分类设计过程是一个“**自底向上，逐步抽象**”的过程。从一个个具体的客观对象可抽象出小类，从小类可以抽象出更大的类。例如长方形水池类可进一步抽象出更宽泛的长方形类，圆形水池类也可以抽象出更宽泛的圆形类。长方形类和圆形类还可以再抽象出几何形状类。越往上，类就越宽泛、越抽象。

Java语言支持面向对象程序设计，用**类**(class)的语法形式来描述类图。**定义**类，就是要描述清楚该类包含哪些**属性**成员(称为**字段**)、**方法**成员(称为**方法**)以及各成员的**访问权限**，定义时使用关键字**class**。例如，程序员使用Java语言来定义长方形鱼池类RectPool和圆形水池类CirclePool，其示意代码如下。

(1) 使用Java语言定义长方形鱼池类RectPool。

```
class RectPool {                          //定义类中包含哪些成员以及各成员的权限
    public double a, b;                   //字段:长度 a、宽度 b
    public double RectCost()              //方法:计算长方形鱼池造价的函数 RectCost()
    {  return  (a * b * 10);  }
}
```

(2) 使用Java语言定义圆形水池类CirclePool。

```
class CirclePool {                        //定义类中有哪些成员以及各成员的权限
    public double r;                      //字段:半径
    public double CircleCost()            //方法:计算圆形水池造价的函数 CircleCost()
    {  return  (3.14 * r * r * 10);  }
}
```

类图相当于**设计时**描述客观对象数据模型的**图纸**,而**类定义**则是**编程时**描述该数据模型的 **Java 代码**。

3.2.3 组装

在设计好类之后,面向对象程序设计进入最后的程序组装阶段。组装程序就是编写主方法,先生产程序零件,然后将它们组装在一起,最终形成完整的程序产品。

类相当于是描述客观事物(即客观对象)数据模型的图纸。可以按照图纸来生产产品,所生产出的产品称为图纸的**实例**。在面向对象程序设计中,类被看作是一种由程序员定义的高级**数据类型**,用类所定义的变量称为**对象**。使用类这种数据类型来定义对象,相当于按照类图纸来生产对象,因此对象也被称为是类的实例,定义对象被称为对类的实例化。

1. 用类定义对象

例如下面的定义对象语句,使用圆形水池类 CirclePool 定义一个圆形清水池对象 cObj1 和一个圆形污水池对象 cObj2。

```
CirclePool    cObj1, cObj2;        //先定义两个 Circle 类的引用变量,分别代表清水池和污水池
cObj1 = new CirclePool();          //在内存中创建一个对象(代表清水池),让 cObj1 引用该对象
cObj2 = new CirclePool();          //在内存中再创建一个对象(代表污水池),让 cObj2 引用该对象
```

计算机执行上述定义对象语句,将按照类图纸在**内存**中创建(即生产)对象 cObj1 和 cObj2,为它们分配内存。这种在内存中创建的对象被称为**内存对象**。图 3-6 以圆形清水池和圆形污水池为例,具体演示了面向对象程序设计从客观对象到内存对象的设计过程。

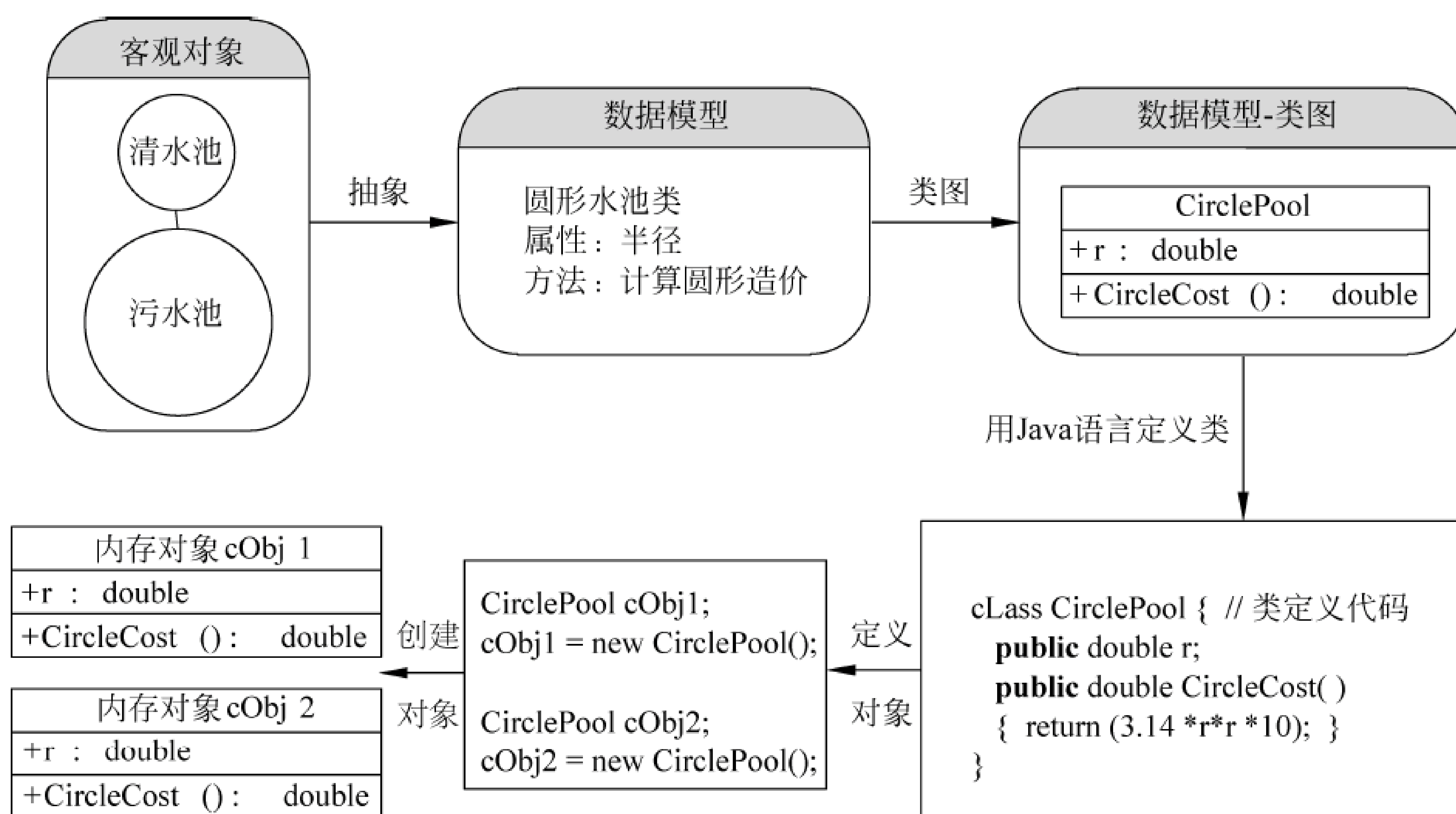


图 3-6 从客观对象到内存对象的设计过程

从图 3-6 可以看出,内存对象 cObj1、cObj2 分别对应的是客观对象“清水池”和“污水池”,它们是客观对象经过**抽象**后在内存中的表现形式。内存对象是用类定义出来的变量,它们具有类所规定的数据成员、函数成员及访问权限。在理解了内存对象和客观对象的概

念之后，请读者记住：后续章节提到程序中的对象通常指代的都是内存对象。

2. 访问对象

用类定义出对象之后，程序员可以访问对象的公有成员。**公有成员**是对象对外开放的操作接口。访问公有成员就是通过接口来操作内存中的对象，例如访问公有属性成员来读写对象的数据，或调用公有方法成员来处理数据。例如：

```
//访问清水池对象 cObj1
cObj1.r = sc.nextDouble();              //输入 cObj1 的半径
totalCost += cObj1.CircleCost();        //计算 cObj1 的造价，并汇总到 totalCost 上
//访问污水池对象 cObj2
cObj2.r = sc.nextDouble();              //输入 cObj2 的半径
totalCost += cObj2.CircleCost();        //计算 cObj2 的造价，也汇总到 totalCost 上
```

假设测算养鱼池工程总造价程序由甲、乙两位程序员分工协作，共同编写。程序员乙负责定义类，编写长方形鱼池类 RectPool 和圆形水池类 CirclePool 的定义代码。程序员甲负责编写主类和主方法，使用上述两个类定义对象（即生产零件），然后访问对象的公有成员（即组装零件），最终完成测算养鱼池工程总造价的功能。例 3-7 给出了完整的测算养鱼池工程总造价的 Java 程序代码。

例 3-7 测算养鱼池工程总造价的 Java 程序代码（面向对象程序设计方法）

程序员甲：主类 + 主方法（Pool.Java）

```
1   import java.util.Scanner;//导入外部程序 Scanner
2
3   public class Pool {                          //主类
4     public static void main(String[] args) {//主方法
5       Scanner sc = new Scanner( System.in );
6       double totalCost = 0;  //保存总造价的变量
7       //处理长方形养鱼池
8       RectPool  rObj;          //定义引用
9       rObj = new RectPool();  //创建长方形鱼池对象
10      rObj.a = sc.nextDouble();    //输入长宽值
11      rObj.b = sc.nextDouble();
12      totalCost += rObj.RectCost(); //汇总造价
13      //处理清水池和污水池
14      CirclePool  cObj1, cObj2;//定义引用
15      cObj1 = new CirclePool();//创建清水池对象
16      cObj2 = new CirclePool();//创建污水池对象
17      cObj1.r = sc.nextDouble();//输入清水池半径
18      cObj2.r = sc.nextDouble();//输入污水池半径
19      totalCost += cObj1.CircleCost();//汇总造价
20      totalCost += cObj2.CircleCost();//汇总造价
21      //显示总造价 totalCost
22      System.out.println( totalCost );
23    }
24  }
```

程序员乙：长方形养鱼池类（RectPool.java）

```
public class RectPool {
   public double a, b;   //字段:长宽
   public double RectCost()
    {  return (a * b * 10);  }
}
```

程序员乙：圆形水池类（CirclePool.java）

```
public class CirclePool {
     public double r;    //字段:半径
     public double CircleCost()
    { return (3.14 * r * r * 10); }
}
```

从例 3-7 中程序员甲所编写的主方法代码可以看出,使用类的程序员在编写程序时应该先用类定义对象,然后再通过对象访问其公有成员,最终完成所需要的程序功能。

学习面向对象程序设计方法,必须从代码分类管理、数据类型、归纳抽象和代码重用等多个维度才能准确理解其思想内涵。本节通过测算养鱼池工程总造价程序的例子,简单介绍了面向对象程序的设计过程,相信读者已经认识了面向对象程序设计方法的概貌。如果想进一步了解面向对象程序设计方法,请阅读面向对象软件工程,或统一建模语言等方面的参考书。从下一节开始,将具体学习 Java 语言中类与对象的语法细则。

本节习题

1. 下列关于类的描述中,错误的是(　　)。
   A. 类是描述客观事物的数据模型　　B. 可以用流程图来描述类的设计
   C. 类的数据成员也被称作属性　　D. 类的函数成员也被称作方法
2. 按照面向对象程序设计的观点,下列关于对象描述中错误的是(　　)。
   A. 客观世界中的事物被称作客观对象
   B. 类是描述客观对象的数据模型
   C. 程序中用类定义出的对象被称作内存对象
   D. 同一个类所定义出的两个内存对象可以有不同的成员
3. 关于面向对象程序设计方法,下列描述中错误的是(　　)。
   A. 面向对象程序设计方法中的类是客观事物抽象后的数据模型
   B. 面向对象程序设计方法更便于代码分类管理
   C. 面向对象程序设计方法所设计出的类代码不能重用
   D. 面向对象程序设计方法是当今程序设计的主流方法
4. 假设编写一个教务管理系统,通过分析可抽象出若干类,其中不应当包括(　　)。
   A. 学生类　　B. 教师类
   C. 课程类　　D. 宿舍类
5. 如果将客观世界中的钟表抽象成一个钟表类,其成员中不应当包含(　　)。
   A. 时、分、秒　　B. 功率　　C. 设置时间　　D. 显示时间

3.3 类与对象的语法细则

Java 语言支持面向对象程序设计方法,为类与对象编程提供了完善的语法规则。

3.3.1 类的定义

定义一个 Java 类,就是用 Java 语言描述该类包含了哪些字段成员、方法成员以及各成员的访问权限。

Java 语法：**定义类**

```
[public] class 类名 {
    [访问权限]  数据类型字段名 [ = 初始值];
    …
    [访问权限] 返回值类型 方法名(形式参数列表) {
        方法体
    }
    …
}
```

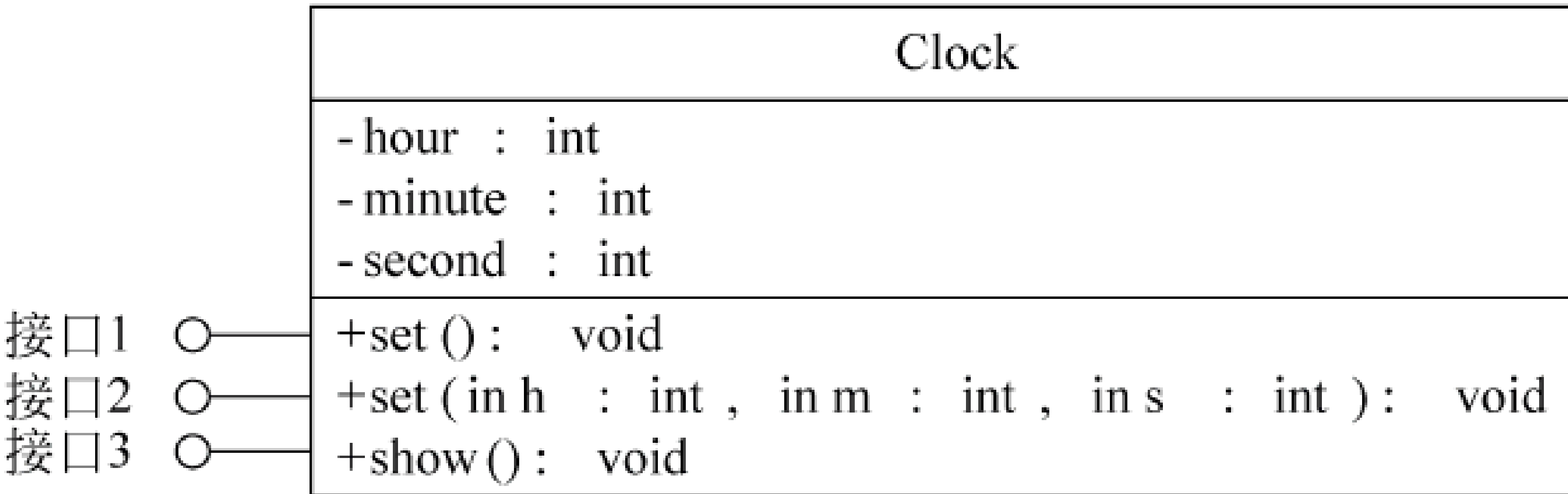
语法说明：

- 定义类时使用关键字 **class**。通常在 class 之前使用关键字 **public** 将类的访问权限设定为公有，也可以省略（此时类将具有默认的访问权限）。**注：** Java 语法的中括号“[…]”表示其中的内容可省略，以下同。
- **类名**需符合标识符的命名规则，习惯上以大写字母开头。
- 类的下属成员有两种，分别是**字段**（存储数据）和**方法**（处理数据的算法）。某些特殊的类可能只包含一种成员，如只包含字段，或只包含方法。
- 类成员的**访问权限**有 4 种，分别是公有权限（**public**）、保护权限（**protected**）、私有权限（**private**）或默认权限（未指定访问权限）。
- 方法成员可以访问本类中任意位置的字段（字段相当于是类中的全局变量），或调用本类中任意位置的其他方法。类成员之间互相访问不需要“先定义，后访问”，也不受权限约束。

例如，可以对客观世界中的钟表进行抽象，用类图描述钟表类的数据模型（见图 3-7）。图 3-7(b)中的钟表类 Clock 包含时、分、秒等 3 个时间属性，将它们设为私有权限 private，这相当于将时针、分针、秒针隐藏（封装）起来。钟表类 Clock 还提供了设置和显示时间的方法，并将它们设为公有权限 public，它们是类对外开放的接口。外界通过接口可以**间接**操作钟表类 Clock 里的私有成员，例如设置或显示被隐藏的时、分、秒数据。



(a) 钟表



(b) 用类图描述钟表类的数据模型

图 3-7　钟表及其数据模型

使用 Java 语言定义钟表类 Clock，将属性定义成字段，将方法定义成函数。其示意代码如例 3-8 所示。

例 3-8 一个钟表类 Clock 的 Java 示意代码(Clock.java)

```
1  import java.util.Scanner;                         //导入外部程序 Scanner
2
3  public class Clock{                               //定义钟表类 Clock
4      //将字段设为 private 权限,即私有成员
5      private  int hour;                            //字段 hour:保存小时数
6      private  int minute;                          //字段 minute:保存分钟数
7      private  int second;                          //字段 second:保存秒数
8      //将方法设为 public 权限,即公有成员
9      public  void set() {                          //不带参数的方法 set():从键盘输入时间
10         Scanner sc = new Scanner( System.in );//创建键盘扫描器对象 sc
11         hour = sc.nextInt();  minute = sc.nextInt();  second = sc.nextInt();
12     }
13     public  void set(int h, int m, int s) {     //带参数的方法 set():用参数设定时间
14         hour = h;  minute = m;  second = s; //将参数分别赋值给对应的字段
15     }
16     public  void show() {                         //方法 show:显示时间
17         System.out.println( hour + ":" + minute + ":" + second );    //时:分:秒
18     }
19 }
```

1. 字段成员的语法细则

(1) **字段**(field)是类中的变量,用于保存数据。

(2) 字段之间的数据类型可以相同,也可以不同。

(3) 字段由程序员命名,需符合标识符的命名规则,习惯上以小写字母开头。字段不能与其他类成员重名。

(4) 定义字段的语法形式类似于定义变量,定义时也可以初始化(**注**:C++语言定义数据成员时不能初始化)。例如:

```
private  int hour = 0;               //将钟表的时间初始化为 0:0:0
private  int minute = 0;
private  int second = 0;
```

(5) 未初始化的字段将被自动初始化成**空值**(即默认值)。表 3-1 列出了不同数据类型所对应的空值。

表 3-1 不同数据类型的空值

类　型	byte/short/int/long	float/double	char	boolean	引用变量
字段默认值	0	0.0	'\u0000'	false	null

(6) 可以使用关键字 **final**(通常放在访问权限之后,数据类型之前)将字段定义成只读字段。只读字段在初始化后,其数值不能修改,例如不允许再次赋值。例如:

```
private final int SECOND = 0;                  //将字段 SECOND 定义成只读字段
```

类中只读字段的作用就相当于是一个常量。只读字段习惯上以大写字母命名。

2. 方法成员的语法细则

(1) **方法**(method)是类中的**函数**(function),其功能通常是对本类中的字段(即数据)进行处理。

(2) 方法所描述的是某种数据处理算法,其中包括 4 大要素。

- **方法名**。方法名是方法的标识,由程序员命名,需符合标识符的命名规则,习惯上以小写字母开头。
- **形式参数列表**。形式参数列表定义了接收输入参数所需的变量,这些变量称为**形式参数**,简称为**形参**。可以有多个形参,每个形参以"数据类型 变量名"的形式定义,形参之间用逗号","隔开。某些方法可能不需要输入参数,此时形式参数列表省略为空。
- **方法体**。方法体是描述数据处理算法的 Java 语句序列,用大括号"{ }"括起来。方法体中可以定义专供本方法使用的变量(称为**局部变量**)。如果方法有**返回值**,则应使用 return 语句返回。返回值的数据类型应与下面的返回值类型一致。
- **返回值类型**。返回值类型也称方法类型,指定了返回值的数据类型。无返回值时应将返回值类型定义为 **void**。void 是 Java 语言的关键字。

方法头"返回值类型 方法名(形式参数列表)"被称为方法的**签名**(signature),它定义了方法的调用接口,即方法名称、输入参数和返回值类型。

(3) 方法可直接访问本类中的任意字段。字段相当于是类中的全局变量。

(4) 方法可以直接调用本类中的任意其他方法。

(5) **重载**(overload)**方法**。类中的两个方法通常不应该重名,但如果它们的功能类似,并且形参的个数不同或数据类型不同,那么这两个方法就可以重名。重名的方法被称为重载方法。例如,下列两个重载的 set()方法分别提供了不同的设置时间方法。

```
public  void set() {                             //不带参数的方法 set():从键盘输入时间
    Scanner sc = new Scanner( System.in );       //创建键盘扫描器对象 sc
    hour = sc.nextInt();  minute = sc.nextInt();  second = sc.nextInt();
}
public  void set(int h, int m, int s) {          //带参数的方法 set():用参数设定时间
    hour = h;  minute = m;  second = s;          //将参数分别赋值给对应的时、分、秒字段
}
```

调用同名的重载方法时,会调用哪个方法呢?编译源程序时,由编译器根据调用语句中实参的个数和类型自动调用形参匹配的那个重载方法(即形参-实参匹配原则)。

3. 类成员访问权限的语法细则

(1) 每个类成员都有并且只有一种访问权限。

(2) 类成员的访问权限有 4 种,它们分别是:

- 公有权限 **public**。被赋予公有权限的类成员是开放的,称为公有成员。
- 私有权限 **private**。被赋予私有权限的类成员将被隐藏,称为私有成员。

- 保护权限 **protected**。被赋予保护权限的类成员是半开放的，称为保护成员。
- 默认权限(未指定访问权限)。被赋予默认权限的类成员也是半开放的。

(3) 定义类时，不同权限成员可以按任意次序编排。为便于阅读，通常将相同权限的成员编排在一起；或将字段编排在一起，将方法编排在一起。

(4) 同类成员之间互相访问，不受权限控制。

(5) 通常，一个类应包含公有权限的成员，否则该类没有对外的接口，无法使用。

3.3.2 对象的定义与访问

和 int、double 等基本数据类型相比，类是一种程序员自己定义的新的数据类型，可称为**类类型**。使用类，就是用类来定义变量，然后访问其下属成员，从而实现类所规定的程序功能。用类所定义的变量被称为是该类的一个对象(或实例)。本节以 3.3.1 节定义的钟表类 Clock 为例，具体讲解对象的定义和访问语法。

1. 定义对象

用类定义(或称创建)对象，就是使用运算符 **new** 来为对象动态分配内存。运算符 new 是 Java 语言的关键字。

运算符 new 能返回所创建对象的**引用**(reference)。“引用”可理解为是指向对象所分配内存单元的“首地址”。可以定义保存引用的变量，这种变量被称为**引用变量**(类似于 C/C++语言里的指针变量)。后续程序将通过引用变量访问对象及其下属成员。引用变量的类型应当与被引用对象的类型一致。例如，

```
Clock obj1;              //预先定义一个 Clock 类型的引用变量 obj1
                         //此时 obj1 的引用值为 null,即还未引用任何对象
obj1 = new Clock();      //创建一个 Clock 类型的对象,将运算符 new 返回的引用保存到 obj1 中
```

上述两条语句可简写为一条语句：

```
Clock obj1 = new Clock(); //定义引用变量的同时创建对象.请注意,"="两边的类型应当一致
```

上述 Java 语句创建一个 Clock 类的对象，并将其引用保存到引用变量 obj1 中。这时，称“引用变量 obj1 **引用了**(或指向了)一个 Clock 对象”，或简单地说“obj1 是一个 Clock 对象”(此时 obj1 被当成了**对象名**)。

类就像是一张图纸，创建对象就是按照图纸在内存中创建一个该图纸的**实例**(instance)，因此创建对象也被称为是对类的**实例化**。计算机会严格按照类定义创建对象，所创建的对象具有类所规定的字段成员、方法成员及访问权限。

和 int、double 等基本数据类型的变量相比，类类型的对象是一种复杂变量，其中包含多个下属成员。按照钟表类 Clock 的定义，其所定义的钟表对象 obj1 将包含 3 个私有的字段成员，即保存时、分、秒数据的 hour、minute 和 second；另外还有 3 个公有的方法成员，即两个重载的设置时间方法 set()和一个显示时间的方法 show()，总共有 6 个下属成员。

2. 访问对象

定义好的对象可以访问。访问对象就是通过其**公有成员**(接口)操作内存中的对象,实现特定的程序功能,如读写对象中的公有字段,或调用其中的公有方法。对象中的公有成员可以访问,非公有成员(即私有/保护/默认权限的成员)只能在特定范围内才能访问,这就是访问对象下属成员时的权限控制。

访问对象下属成员需使用**成员运算符“.”**,以“**对象名.字段名**”的形式访问对象中的字段,或以“**对象名.方法名(实参列表)**”的形式调用对象中的方法。

在定义了钟表对象 obj1 之后,可以访问其公有成员实现钟表的功能。例如,先设置钟表对象 obj1 的时间,然后再显示其时间。

(1) 访问钟表对象的公有成员。

```
Clock obj1 = new Clock();          //创建一个钟表对象 obj1
obj1.set();                        //调用对象 obj1 的公有方法 set(),输入时、分、秒数据
obj1.show();                       //调用对象 obj1 的公有方法 show(),显示其时间
```

(2) 可以用钟表类 Clock 定义(生产)多个钟表对象。

```
Clock obj2 = new Clock();          //创建第二个钟表对象 obj2
obj2.set( 8, 30, 15 );             //调用对象 obj2 的公有方法 set(),设置时间 8:30:15
obj2.show();                       //调用对象 obj2 的公有方法 show(),显示其时间
```

(3) 一个对象可以被多次引用。

```
Clock obj ;                        //再定义一个 Clock 类的引用变量 obj
obj = obj1;                        //赋值后,obj 与 obj1 引用同一个对象,该对象被引用了 2 次
obj.set( 12, 0, 0);                //通过引用变量 obj 操作钟表对象,将其时间设为 12:0:0
obj.show();                        //显示对象的时间,显示结果应为 12:0:0
obj1.show();                       //显示 obj1 所引用对象的时间,显示结果也为 12:0:0
```

在这个例子中,引用变量 obj 和 obj1 引用了同一个钟表对象,称该钟表对象**被引用的次数**为 2。

3.3.3 引用数据类型

Java 语言中的数据类型可分为两大类。

1) 基本数据类型

基本数据类型(primitive data type)有 byte、short、int、long、float、double、char、boolean,共 8 种。

2) 引用数据类型

所有的**类**类型、**数组**类型、**接口**类型、**枚举**类型等被统称为引用数据类型(reference data type)。

注:数组、接口和枚举类型将在后续章节讲解。

例 3-9 给出 Java 语言中这两大类数据类型的应用示例。学习过 C 语言的读者可以对照 C 语言来帮助理解。

例 3-9 Java 语言中的数据类型与 C 语言中的数据类型

	Java 语言	C 语言

基本数据类型	int x; //基本数据类型的变量 x
	x = 10; //通过变量名访问内存单元
	
	//基本数据类型**没有**指针的概念
	//基本数据类型**不能**动态分配内存
		int x; //基本数据类型的变量 x
		x = 10; //通过变量名访问内存单元
		
		//指针变量与动态分配内存
		int * **p** = (int *) **malloc**(sizeof(int));
		* **p** = 10; //通过指针变量访问内存单元
		free(**p**); //释放动态分配的内存

自定义数据类型	class **Rectangle**{ //长方形类
	    public double a, b; //字段:长、宽
	}
	
	//类类型的引用变量与动态分配
	Rectangle rect; //定义引用变量
	rect = **new** Rectangle(); //动态分配
	
	//通过引用变量 rect 访问下属成员
	rect.**a** = 5; rect.**b** = 10;
	/*
	Java 语言中的类类型必须动态分配,引用变量**融合**了 C 语言中**指针变量**和**变量名**的功能.
	*/
		struct **Rectangle** { //长方形结构体类型
		    double a, b; //成员:长、宽
		};
		//定义结构体类型的变量
		struct **Rectangle** rect; //定义结构体变量
		//通过变量名访问下属成员
		rect.**a** = 5; rect.**b** = 10;
		//动态分配结构体变量
		struct Rectangle * **p**; //定义指针变量
		p = (struct Rectangle *) **malloc**(
		sizeof(struct Rectangle)); //动态分配
		//通过指针变量访问下属成员
		(* p).**a** = 5; (* p).**b** = 10;
		或
		p -> **a** = 5; p -> **b** = 10;
		free(**p**);

1. Java 语言的基本数据类型

Java 语言中,使用基本数据类型定义变量,定义时直接为变量分配内存单元。后续程序将通过变量名访问变量的内存单元。

与 C 语言对照理解:上述使用基本数据类型的方法,Java 语言与 C 语言基本一样。但 C 语言还可以通过函数 malloc()动态分配内存,并将所分配内存的首地址保存到指针变量中,后续程序通过指针变量间接访问所分配的内存单元。

Java 语言中,基本数据类型没有指针的概念,也不能动态分配内存。

2. Java 语言的引用数据类型

Java 语言中,使用引用数据类型(例如类类型)所定义的变量被改称为对象。使用引用数据类型定义对象,需分两步完成。

(1) 先定义引用数据类型的引用变量。

(2) 再用运算符 new **创建**引用数据类型的**对象**,并将所返回的对象引用保存到引用变量中,后续程序将通过引用变量访问对象及其下属成员。**注**:使用运算符 new 创建对象,实际上是为对象动态分配内存,所返回的引用可理解为是所分配内存单元的首地址。

与 C 语言对照理解:Java 语言里的引用类型,相当于是 C 语言里的自定义数据类型。例如,可以将 Java 语言的类类型与 C 语言里的结构体类型做类比。

C 语言中，可以使用结构体类型直接定义出结构体变量，然后通过变量名访问其下属成员；也可以为结构体变量定义指针变量并动态分配内存，然后通过指针变量间接访问其下属成员。

Java 语言中，使用类类型不能直接定义对象，只能为对象定义引用变量并使用运算符 new 为对象动态分配内存，然后通过引用变量间接访问对象及其下属成员。Java 语言里的**引用变量**，在本质上类似于指向对象的**指针变量**，而其使用形式又类似于**对象名**。

3. 变量及对象的内存分配

这里通过图 3-8 所示的内存分配示意图来直观讲解基本数据类型的普通变量、引用数据类型的引用变量和对象是如何分配内存的。

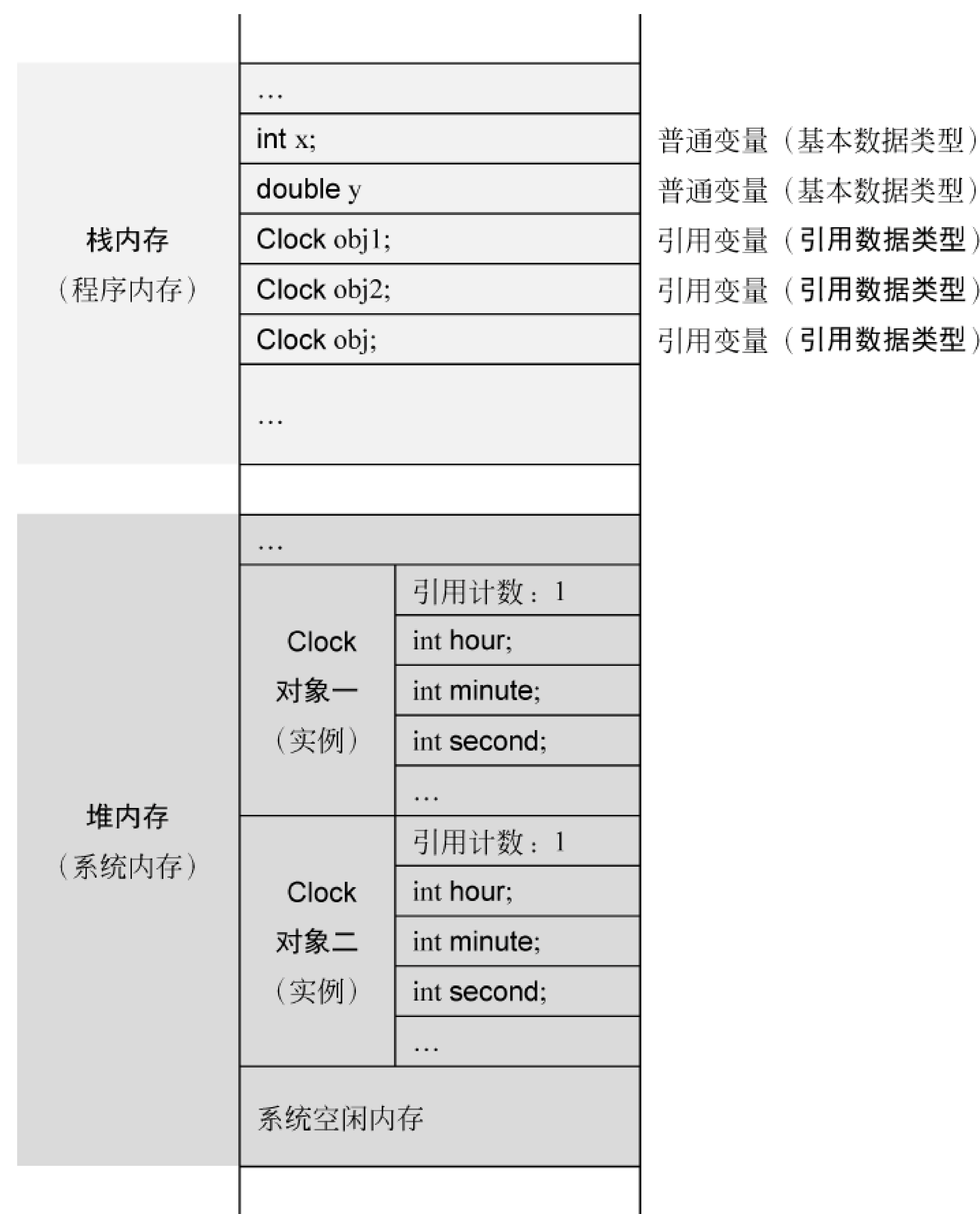


图 3-8　变量和对象的内存分配示意图

1）变量

Java 语言中，使用基本数据类型定义变量，计算机在执行其定义语句时即为变量分配内存单元。例如：

```
int    x;               //为 int 型变量 x 分配 4 字节的内存单元
double   y;             //为 double 型变量 y 分配 8 字节的内存单元
```

为便于讲解,图 3-8 将基本数据类型的变量 x、y 称作**普通变量**,这样可以与引用数据类型的引用变量区分开来。

使用引用数据类型定义**引用变量**,计算机在执行其定义语句时也立即为引用变量分配内存单元。例如:

```
Clock obj1;                //定义一个 Clock 类型的引用变量 obj1
```

计算机会为引用变量 obj1 分配几字节呢? 从**概念**上,可以将引用变量理解成保存地址的指针变量。通常,32 位操作系统中的内存地址为 32 位(4 字节),64 位操作系统中的内存地址为 64 位(8 字节)。

Java 语言中,在方法体内部定义的普通变量和引用变量都属于**局部变量**,只能在方法体内部访问。执行 Java 程序,当执行到方法体中的局部变量定义语句时,Java 虚拟机将在为程序预留的**栈**(stack)**内存**中为局部变量分配内存单元。

2) 对象

使用引用数据类型定义引用变量,这时计算机还没有创建对象,即对象还没有分配内存。创建对象必须使用运算符 new 来动态分配内存。Java 虚拟机负责管理计算机系统的空闲内存,这些内存被称为**堆**(heap)**内存**。对象的内存单元是在堆内存中动态分配的(图 3-8)。例如:

```
obj1 = new Clock();     //动态创建一个 Clock 类型的对象一,并将其引用保存到引用变量 obj1 中
Clock obj2 = new Clock();   //再创建一个 Clock 类型的对象二,将其引用保存到引用变量 obj2 中
```

运算符 new 创建对象后,将返回该对象的引用。将引用保存到某个引用变量中,称对象被**引用**了一次。例如,将 Clock 类型对象一的引用保存到引用变量 obj1 中,则对象一被 obj1 引用了一次。同样,对象二也被 obj2 引用了一次。

一个对象可以被引用多次,即被多个引用变量同时引用。Java 虚拟机内部为每个对象都设置了一个**引用计数器**,用于统计对象被引用的次数。例如:

```
Clock obj;            //再定义一个 Clock 类型的引用变量 obj
obj = obj1;           //将 obj1 赋值给 obj,则 obj 与 obj1 引用了同一个对象(即对象一)
```

此时对象一被引用了两次,其引用计数器加 1,变成 2。引用变量可以改变引用而引用别的对象。例如:

```
obj = obj2;     //将 obj2 赋值给 obj,则 obj 改变了引用,现在与 obj2 引用同一个对象(即对象二)
```

此时对象一减少一次引用,其引用计数器降为 1;而对象二则增加一次引用,其引用计数器升为 2。

将引用变量赋值为 **null**(空引用),则引用变量不引用任何对象。例如:

```
obj = null;           //此时,obj 不引用任何对象,对象二的引用计数器相应地被降为 1
```

这时如果继续将 obj2 也设为空引用。例如:

```
obj2 = null;          //此时,obj2 也不引用任何对象,对象二的引用计数器相应地被降为 0
```

当一个对象不被任何变量所引用时,其引用计数器为 0,这意味着该对象不再有任何使

用价值了。Java 虚拟机将引用计数器为 0 的对象称作是内存中的“**垃圾**”，是可以被回收的对象。Java 虚拟机会自动在后台定期删除引用计数器为 0 的对象，收回其内存空间，以便给其他对象继续使用，这就是 Java 语言中的**垃圾回收**（garbage collection）机制。

因为 Java 语言具有自动的垃圾回收机制，程序员只需要考虑如何创建和使用对象。在使用完之后，程序员不需要再花精力去考虑如何删除对象，释放其内存空间，这项工作将由 Java 虚拟机帮程序员自动完成。

3.3.4　3 种不同的变量

从定义位置和功能上划分，Java 语言中的变量可分为 3 种。

（1）**字段**。定义在类中的变量成员，用于保存属性数据。字段相当于是类中的全局变量，可以被类中的所有方法成员访问。

（2）**局部变量**。类中方法成员在方法体内部定义的变量，仅能在所定义的方法体中访问。

（3）**形式参数（形参）**。类中方法成员在头部小括号里面定义的变量，用于接收原始数据。形参仅能在所定义的方法体中访问。

注：*Java 语言没有全局变量的概念。*

字段、局部变量或形参都可以是基本数据类型，也可以是引用数据类型。下面通过一个为钟表设置整点时间（即分钟和秒数为 0）的方法 setHour()来具体讨论一下这 3 种变量，参见例 3-10。

例 3-10　一个为钟表设置整点时间的 Java 示例代码（ClockTest.java）

```
1  import java.util.Scanner;                      //导入外部程序 Scanner
2
3  public class ClockTest {                       //主类
4      public static void main(String[] args) {  //主方法
5          int hour;                              //局部变量：普通变量，未初始化
6          Clock c1;                              //局部变量：引用变量，未初始化
7          hour = 12;    c1 = new Clock();        //为局部变量赋值
8          c1.set( 8, 30, 15 );                   //设置钟表对象 c1 的时间
9          c1.show();                             //显示结果 8:30:15
10
11         Clock c2;                              //再定义一个局部引用变量 c2
12         c2 = setHour( c1, hour );  //调用 setHour()将 c1 设为整点，并将返回值赋值给 c2
13         c2.show();                 //显示 c2 的时间，结果应为 12:0:0
14         c1.show();                 //显示 c1 的时间，结果也为 12:0:0
15     }
16
17     private static Clock setHour( Clock rc, int h ) {        //将钟表时间设为整点的方法
18         rc.set( h, 0, 0 );       //设置 rc 的时间，小时数为接收的参数 h，分钟和秒数设为 0
19         return rc;
20     }
21 }
```

例 3-10 程序的代码说明如下。

(1) 方法中定义的变量是局部变量。

代码第 5 行定义的 hour 是基本数据类型 int 的普通变量，代码第 6 行定义的 c1 是引用数据类型 Clock 的引用变量，它们都属于局部变量。这两个变量在定义都没有初始化，代码第 7 行对它们进行了赋值。

(2) 调用对象的方法成员来处理对象。

引用变量 c1 引用了一个钟表对象，可以调用其下属的方法成员 set()和 show()来设置、显示时间，例如代码第 8～9 行。调用方法 set()和 show()，就是调用对象下属的方法成员来处理对象。

(3) 调用外部方法来处理对象。

代码第 12 行调用方法 setHour()，将钟表对象 c1 的时间设为整点时间。方法 setHour()是主类定义的方法成员，它不是钟表对象下属的方法成员。调用方法 setHour()来设置钟表对象 c1 的时间，就是调用外部方法来处理对象。

(4) 调用方法时的形实结合。

代码第 12 行调用方法 setHour()，其中的实参 c1 是引用数据类型的变量，用于传递一个钟表对象；实参 hour 是基本数据类型的变量，用于传递一个表示小时数的数值。传递数据时，实参将按位置顺序一一对应地传递给方法中的形参，这就是调用方法时参数的**形实结合**。例如，代码第 12 行调用方法 setHour()时，会按位置顺序将第一个实参 c1 传递给方法中的形参 rc，将第二个实参 hour 传递给形参 h。

Java 语言中，方法间传递基本数据类型数据时直接传递数值，即**值传递**；而传递引用数据类型的对象时所传递的是对象引用(不是对象本身)，称为**引用传递**。换句话说，Java 语言传递基本数据类型数据时统一采用值传递，而传递引用数据类型的对象时统一采用引用传递。

(5) 引用传递时形参和实参所引用的是同一个对象。

代码第 12 行调用方法 setHour()，形参 rc 接收到实参 c1 所保存的对象引用，此时 rc 将引用 c1 所指向的钟表对象，即形参和实参引用的是同一个对象。这时，被引用钟表对象的计数器将增加到 2。在方法 setHour()中设置 rc 的时间，实际上就是设置实参 c1 的时间，因为它们引用的是同一个钟表对象。

(6) 调用方法时的返回值。

方法 setHour()有一个返回值，即代码第 19 行 return 语句中的形参 rc，它是一个 Clock 类型的对象引用。Java 语言中，当返回值是基本数据类型时直接返回数值；是引用数据类型时返回的则是对象的引用(不是对象本身)。

代码第 12 行在调用方法后将其返回值赋值给另一个引用变量 c2，此时 c2 和形参 rc、实参 c1 都引用了同一个钟表对象。**注**：当方法 setHour()执行完退出时，形参 rc 将被自动删除，钟表对象的引用计数器会先减一，赋值给 c2 后再加一，计数结果为 2。

当执行到主方法中代码第 13 行时，引用变量 c2 和 c1 引用的是同一个钟表对象。该钟表对象的时间被方法 setHour()设置为 12:0:0。分别调用 c2、c1 的方法 show()，所显示出的时间将是一样的，都是 12:00:00。

3.3.5 类与对象的编译原理

1. 类代码的编译

编译时,Java 编译器将源程序编译成等效的字节码程序。当编译类中的方法成员时,编译器会对其定义代码做出某些调整,然后再进行编译。例如,编译器在编译钟表类 Clock 中的方法成员 set(int h, int m, int s)和 show()时会做出如下调整(见例 3-11)。

例 3-11 钟表类 Clock 中方法成员的编译举例

```
方法成员 set(int h, int m, int s): 调整前
1  public  void set( int h, int m, int s ) {
2    hour = h;
3    minute = m;
4    second = s;
5  }
方法成员 show(): 调整前
1  public  void show() {
2   System.out.println(
3    hour + ":" + minute + ":" + second );
4  }
```

```
方法成员 set(int h, int m, int s): 调整后
public  void set( Clock this, int h, int m, int s ) {
  this.hour = h;
  this.minute = m;
  this.second = s;
}
方法成员 Show(): 调整后
public  void show( Clock this )
 System.out.println(
  this.hour + ":" + this.minute + ":" + this.second );
}
```

从例 3-11 可以看出,编译器在编译类中方法成员时会对其定义代码做出如下两点调整。

(1) 添加一个本类的对象引用 this,作为方法的第一个形参。例如:

```
Clock this
```

(2) 修改方法体中所有对本类成员的访问形式,在成员名之前加"**this.**"。**例如**:

```
this.hour、this.minute、this.second
```

这是一种通过引用 this 访问对象下级成员的形式。编译器为什么要对类中方法成员做这样的调整呢? 我们将在下面再做解释。这里请读者先记住:形参 this 是一个引用,在方法被调用时将会引用某个具体的对象。

2. 调用对象方法成员语句的编译

假设创建一个 Clock 类型的对象,然后调用其方法成员来设置、显示时间。例如:

```
Clock obj1 = new Clock();          //创建钟表对象 obj1
obj1.set( 8, 0, 0 );               //将 obj1 的时间设为 8:0:0
obj1.show();                       //显示 obj1 的时间,显示结果 8:0:0
```

编译器在编译上述两条调用对象方法成员的语句时,会对其调用形式做调整:将方法名前面的对象引用 obj1 移到后面的小括号里,将其作为调用方法时的第一个实参。例如:

```
obj1.set( 8, 0, 0 );               //将 obj1 的时间设为 8:0:0
```

将被调整为:

```
set( obj1, 8, 0, 0 );
```

调整后,调用方法"void set(Clock **this**, int h, int m, int s) { … }",实参 obj1 将被传递给方法里的第一个形参 this,然后在方法体中按如下形式设置 this 的时间:

```
this.hour = h;    this.minute = m;    this.second = s;
```

形参 this 和实参 obj1 引用的是同一个钟表对象,因此设置 this 的时间实际上就是设置 obj1 的时间。

调用对象的方法成员,形参 this 所引用的就是当前调用方法的对象,这个对象被称为**当前对象**。同理,下列调用语句:

```
obj1.show();                        //显示 obj1 的时间,显示结果 8:0:0
```

也会被调整为:

```
show( obj1 );
```

执行该语句将显示 obj1 的时间,显示结果为 8:0:0。

3. 同类的多个对象在内存中共用一份方法代码

编译器在编译类中方法成员的定义代码时会增加一个形参 this;在编译调用方法成员的语句时会将对象引用作为实参传递给 this。编译器这么做是为了让同类的多个对象在内存中共用一份方法成员的代码,从而降低内存占用。例如,创建两个 Clock 类型的对象 obj1 和 obj2:

```
Clock obj1 = new Clock();           //创建钟表对象 obj1
Clock obj2 = new Clock();           //创建钟表对象 obj2
```

Clock obj1;		引用对象一
Clock obj2;		引用对象二
对象一(实例)	int hour;	对象一的内存单元
	int minute;	
	int second;	
对象二(实例)	int hour;	对象二的内存单元
	int minute;	
	int second;	

图 3-9 创建两个钟表对象 obj1 和 obj2

执行上述语句,计算机将为对象 obj1 和 obj2 分配内存,如图 3-9 所示。

3.3.2 节曾说过,计算机会严格按照类定义创建对象,所创建的对象具有类所规定的字段成员、方法成员及访问权限。按照 Clock 类的定义,钟表对象应包含 3 个字段成员,即保存时、分、秒数据的 hour、minute 和 second;另外还有 3 个方法成员,即两个重载的设置时间方法 set()和一个显示时间的方法 show(),总共有 6 个成员。

细心的读者可能发现,图 3-9 中的对象 obj1、obj2 都只包含字段成员,但没有包含任何方法成员。换句话说,计算机创建对象时只会为字段成员分配内存,内存中的对象实际上只是一组属性数据。可以简单地说:**对象即数据**。

可为什么能够调用到对象的方法成员,方法成员的调用又是如何实现的呢?例如,调用钟表对象的设置时间方法 set(int h, int m, int s):

```
obj1.set( 8, 0, 0 );                    //将 obj1 的时间设为 8:0:0
obj2.set( 9, 30, 15 );                  //将 obj2 的时间设为 9:30:15
```

定义同类的多个对象，它们都包含字段成员，用于保存各自的属性数据（或称为对象各自的状态）。**理论上**，每个对象所占用的内存空间等于类中全部字段成员所需内存空间的总和。例如，每个钟表对象都有自己的时、分、秒数据，需要为字段（即 hour、minute、second）分配内存来保存这些数据。Clock 类将字段 hour、minute、second 都定义成 int 型（4 字节），因此每个钟表对象所占用的内存空间为 3×4＝12 字节。

但对所有钟表对象来说，它们设置时间的方法一样，显示时间的方法也一样。因此编译器在编译时，通过调整方法成员的定义代码及调用形式，巧妙地实现了执行时程序中的多个同类对象共用方法成员，内存中只需要保存一份方法成员的代码。形参 this 在这中间扮演了重要角色，this 是 Java 语言的关键字。例如：

```
void set(Clock this, int h, int m, int s) { … }  //类代码的编译
set( obj1, 8, 0, 0 );                           //调用对象方法成员语句的编译
set( obj2, 9, 30, 15 );
```

程序员在编写方法成员代码时，形参 this 是隐含的，在方法体中访问其他类成员也不需要添加 this 引用，这些都由编译器在编译时自动添加。程序员可以在访问类成员时显式添加“**this.**”，也可以通过 this 获取**当前对象**的引用，或把 this 作为实参将当前对象的引用继续传递给其他方法。

另外，程序员还可以利用 this 来区分与形参或局部变量重名的字段。例如：

```
public void set( int hour, int minute, int s ) {  //设置时间:形参 hour、minute 与字段重名
    this.hour = hour;                             //this.hour 指代的是字段 hour
    this.minute = minute;                         //this.minute 指代的是字段 minute
    second = s;                                   //形参 s 与字段 second 不重名,可以不用 this
}
```

3.3.6　类的构造方法

程序执行过程中，计算机创建对象，为对象分配内存空间。创建对象的过程称为对象的**构造**（construction）。类就像是一张图纸，构造对象就是按照图纸在内存中创建一个内存对象。Java 语言允许程序员参与对象的构造过程，例如为字段成员赋初始值（相当于是对象的出厂设置），实现创建对象时的初始化。

与基本数据类型的变量相比，类类型的对象可以包含多个字段成员，其构造过程更复杂，涉及的内容更多。为此，面向对象程序设计需要定义专门的构造方法来实现构造的功能。

构造方法（constructor）是类中一种特殊的方法，其主要用途是在创建对象时提供初始化方法。构造方法需遵守如下语法细则。

（1）构造方法的名字必须与类名相同。

（2）构造方法通过形参传递初始值，实现对新建对象字段成员的初始化。

（3）构造方法可以重载，即定义多个同名的构造方法，这样可以提供多种形式的初始化方法。

(4) 构造方法没有返回值,定义时不能写返回值类型,写 void 也不行。

(5) 构造方法通常是类外调用,其访问权限不能设为 private,否则将不能使用运算符 new 来创建该类的对象。只有在少数特殊应用场合(例如单例模式)才会将构造方法设为 private。

从语法形式上讲,一个类必须有构造方法。如果一个类没有定义构造方法,则编译器在编译时将自动添加一个空的构造方法(称为默认构造方法),其语法形式为:

```
类名() { }                                   //其实什么事情也没做
```

1. 初始化对象

例如,使用钟表类 Clock 创建对象时希望能初始化对象的字段成员,为它们赋以不同的值,这样就能创建出具有不同时、分、秒初始值的钟表对象。

初始化对象的方法是先在类中添加构造方法,然后在创建对象时给出初始值。例如,先在类 Clock 的定义代码中添加如下两个重载的构造方法:

```
public Clock( int p1, int p2, int p3 ) {              //带形参的构造方法
    hour = p1;      minute = p2;      second = p3;
}
public Clock() {                                      //不带形参的构造方法
    this( 0, 0, 0 );             //通过 this 调用本类重载的带形参构造方法(须为第一条语句)
    //hour = 0;      minute = 0;      second = 0;     //或直接赋值
}
```

创建对象时,根据实参-形参匹配原则来决定具体调用哪个构造方法。例如:

```
Clock obj1 = new Clock( 8, 30, 15 );                 //给了实参,将调用带形参的构造方法
Clock obj2 = new Clock();                            //未给实参,将调用不带形参的构造方法
Clock obj = obj1;                                    //未实际创建对象,不会调用构造方法
```

构造方法的调用遵循以下 3 条原则。

(1) 构造方法是在使用类创建对象时由系统**自动调用**的(可理解为由 new 运算符调用),程序员不能直接调用构造方法。

(2) 使用类创建对象,每创建一个对象就会调用一次类的构造方法,创建多少个对象就会调用多少次构造方法。

(3) 需要注意的是,定义引用变量时并不会创建对象,因此也不会调用构造方法。

2. 拷贝构造方法

在类 Clock 的定义代码中再添加一个如下的构造方法。

```
public Clock( Clock oldObj ) {             //拷贝构造方法:形参接收一个本类对象的引用
    hour   = oldObj.hour;                  //将形参所引用对象的字段一一对应地复制给新对象
    minute = oldObj.minute;
    second = oldObj.second;
    //注:在本类中可以访问本类对象的私有成员,例如访问 oldObj 的私有字段
}
```

这种形式的构造方法被称为拷贝构造方法，其功能是用一个已经存在的对象来初始化当前正在创建的新对象。例如：

```
Clock obj1 = new Clock( 8, 30, 15 );
Clock obj2 = new Clock( obj1 );          //用已存在的对象 obj1 初始化新建对象 obj2
```

3. 显示对象的创建过程

可以在构造方法的方法体中调用 println()方法来显示某些信息，让程序员实时观察对象的构造过程，这样可以帮助程序员检查程序代码中的错误。例如，为类 Clock 中的 3 个构造方法添加显示信息的语句，用于显示当前哪个构造方法被调用了。

```
public Clock() {                                               //不带形参的构造方法
    hour = 0;  minute = 0;  second = 0;
    System.out.println( " Clock() called." );                 //添加显示信息的语句
}
public Clock( int p1, int p2, int p3 ) {                       //带形参的构造方法
    hour = p1;  minute = p2;  second = p3;
    System.out.println( " Clock(int p1, int p2, int p3) called." ); //添加显示信息的语句
}
public Clock( Clock oldObj ) {                                 //拷贝构造方法
    hour = oldObj.hour;  minute = oldObj.minute;  second = oldObj.second;
    System.out.println( " Clock( Clock oldObj ) called." );   //添加显示信息的语句
}
```

这时执行下列创建对象语句，构造方法会执行其中的 println()语句，这样就能在显示器上实时看到哪个构造方法被调用了。

```
Clock obj1 = new Clock();               //未给实参,将显示 Clock() called.
Clock obj2 = new Clock( 8, 30, 15 );    //给了 3 个 int 型实参,将显示
                                        //Clock(int p1, int p2, int p3) called.
Clock obj3 = new Clock( obj2 );         //给了 1 个 Clock 型引用实参,将显示
                                        //Clock( Clock oldObj ) called.
```

以上提示信息显示出了对象的构造过程，这为程序员检查对象创建错误提供了线索。

3.3.7 类的静态成员

Java 语言是纯面向对象的程序设计语言，程序中没有游离在类外的**全局变量**和**外部函数**。在需要用到全局变量或外部函数的场合，可以将它们定义成类的静态成员。在类中定义静态成员时，需使用关键字 **static** 进行限定。下面通过一个程序实例来具体讲解类中静态成员的应用场景及语法细则。

使用钟表类 Clock 可以创建多个钟表对象。假设希望对所创建的钟表对象进行计数。例 3-12 使用模拟 C/C++语言的伪代码，给出一种使用全局变量和外部函数来实现对象计数的方法。

例 3-12 对钟表类 Clock 进行对象计数的伪代码(模拟 C/C++语言)

```
1  int totalClock = 0;        //模拟 C++语言:定义一个全局变量,记录所创建的 Clock 对象个数
2  void plusObj() {   totalClock ++;   }   //模拟 C/C++语言:定义一个外部函数,将计数加 1
3
4  public class Clock {                    //钟表类 Clock
5      private  int hour, minute, second;  //字段成员
6      public  void set() { … }            //两个设置时间的方法 set(代码省略)
7      public  void set(int h, int m, int s) { … }
8      public  void show() { … }           //显示时间方法 show(代码省略)
9      public  Clock()                     //定义一个构造方法
10     { …;   plusObj();  }                //希望通过构造方法为钟表对象增加计数功能
11 }
```

例 3-12 定义了一个全局变量 totalClock 来记录所创建的 Clock 对象个数。为演示语法,又单独定义了一个将计数加 1 的计数器函数 plusObj()。

使用钟表类 Clock 创建对象时,计算机将自动调用类的构造方法。例 3-12 在构造方法中添加了一条调用计数器函数 plusObj()的语句。使用钟表类 Clock 每创建一个对象,构造方法会被自动执行一次。通过计数器函数将计数变量 totalClock 加 1,这样 totalClock 中就保存了所创建钟表对象的个数。

但 Java 语言不允许在类外定义任何的全局变量或函数。可以将全局变量 totalClock 和计数器函数 plusObj()归属到钟表类 Clock 中,以静态成员的形式来实现与上述对象计数完全相同的功能(见例 3-13)。

例 3-13 通过静态成员实现对钟表类 Clock 进行对象计数的 Java 示意代码(Clock. java)

```
1  public class Clock {                              //定义钟表类 Clock
2      public static  int totalClock = 0; //定义一个静态字段,记录已创建的 Clock 对象个数
3      private static  void plusObj() {  totalClock ++;  }   //定义一个静态方法,将计数加 1
4
5      private int hour, minute, second;             //字段成员
6      public void set() { …… }                      //不带参数的设置时间方法 set(代码省略)
7      public void set(int h, int m, int s) { … }    //带参数的设置时间方法 set(代码省略)
8      public void show() { … }                      //显示时间方法 show(代码省略)
9      public Clock()                                //定义一个构造方法
10     { …;   plusObj();  }                          //通过构造方法为钟表对象增加计数功能
11 }
```

可以使用 3-13 中的钟表类 Clock 创建多个对象。例如:

```
public class ClockTest {                       //测试类
    public static void main(String[] args) {   //主方法
        Clock c1 = new Clock();                //创建第一个对象 c1
        Clock c2 = new Clock();                //创建第二个对象 c2
        //显示 totalClock 中保存的对象个数,下面 3 条语句的显示结果都为 2
        System.out.println( Clock . totalClock ); //通过类名访问静态成员 totalClock
        System.out.println( c1 . totalClock );    //或通过任一对象引用(c1 或 c2)访问静态成员
```

```
            System.out.println( c2 . totalClock );
        }
    }
```

Java 语言中，对于一些需要**共用**的**全局变量**或**外部函数**，可以将它们划归到某个具有关联关系的类中，与类中的其他成员一起进行统一管理。但这些全局变量或外部函数与类中的其他成员是有区别的，Java 语言使用关键字 static 将它们定义成静态成员，分别称为**静态字段**和**静态方法**。

1. 静态字段的内存分配

静态字段单独分配内存，并且只在加载类代码时分配一次。换句话说，不管定义多少个对象，Java 虚拟机只会在第一次使用类时为静态字段单独分配内存单元，所有对象都不包含静态字段。

假设，使用 3-13 中的钟表类 Clock 创建两个对象 c1 和 c2：

```
Clock c1 = new Clock();
Clock c2 = new Clock();
```

执行上述语句，计算机将为对象 c1 和 c2 分配内存，如图 3-10 所示。

Clock c1;		引用对象一
Clock c2;		引用对象二
static　int totalClock;		静态字段（类字段）
对象一（实例）	int hour;	普通字段（实例字段）
	int minute;	
	int second;	
对象二（实例）	int hour;	普通字段（实例字段）
	int minute;	
	int second;	

图 3-10　静态字段与普通字段的区别

图 3-10 中，Java 虚拟机在使用类 Clock 创建第一个对象 c1 时，会加载类 Clock 的定义代码，并为静态字段 totalClock 分配好内存单元，然后再创建对象 c1、c2。这两个对象中只包含普通字段的内存单元，例如 hour、minute、second。

任何对象都可以像访问普通字段一样访问静态字段。例如：

```
c1. totalClock、c2. totalClock
```

看起来，似乎每个对象都包含一个静态字段，但实际所访问的是同一个内存单元。静态字段是被本类所有对象共用的成员，它也因此被称作**类字段**(class field)，可以通过类名直接访问。例如：

```
Clock . totalClock
```

对应地,非静态的普通字段(例如 hour)被称作**实例字段**(instance field),因为只有使用运算符 new 创建的对象实例才会包含普通字段。每个对象实例都包含各自的实例字段,用于保存各自的属性数据,它们占用不同的内存单元。

2. 静态字段的语法细则

1) 关键字 static

在类中定义静态字段需使用关键字 static 进行限定,通常放在访问权限之后,数据类型之前。定义静态字段时可以初始化。

2) 在本类访问静态字段

在本类的方法成员中访问静态字段,直接使用字段名访问,访问时不受权限约束。这一点与访问普通字段是一样的。

3) 不能通过 this 访问静态字段

静态字段是类字段,没有包含在对象中,因此不能使用关键字 this 访问静态字段。这一点与访问普通字段是不一样的。**注**:this 是指向当前对象的引用变量。

4) 在类外访问静态字段

在类外其他方法(例如主方法)中访问静态字段需以"**类名.静态字段名**"的形式访问,或通过任何一个该类对象引用以"**对象引用名.静态字段名**"的形式访问。类外访问静态字段受权限约束。

在类外访问静态字段时可以不创建对象,直接通过类名访问。例如,可以按如下形式直接访问类 Clock 的静态字段 totalClock:

```
Clock. totalClock          //注:最终是否可以访问,还需要看 totalClock 的访问权限是否允许
```

3. 静态方法的语法细则

可以将外部函数划归到某个具有关联关系的类中,作为类的静态方法成员进行管理,或者将专门用于处理静态字段的方法成员定义成静态方法。

1) 关键字 static

在类中定义静态方法需使用关键字 static 进行限定,通常放在访问权限之后,返回值类型之前。

2) 在本类调用静态方法

本类中的所有方法成员都可以调用静态方法。调用时直接使用方法名,并且不受访问权限约束。这一点与调用普通方法是一样的。

3) 在类外调用静态方法

与静态字段一样,静态方法是一种**类方法**,与对象实例无关。对应地,非静态的普通方法只能在创建对象实例之后才能调用,因此被称为**实例方法**。

在类外其他方法(例如主方法)中调用静态方法,需要以"**类名.静态方法名()**"的形式调用,或通过任何一个该类对象引用以"**对象引用名.静态方法名()**"的形式调用。类外调用受访问权限约束。

在类外调用静态方法时可以不创建对象，直接通过类名调用。例如，可以按如下形式直接调用类 Clock 的静态方法 **plusObj**()：

```
Clock.plusObj()            //注：最终是否可以调用，还需要看 plusObj()的访问权限是否允许
```

4）静态方法访问本类其他成员

静态方法只能访问本类中的静态字段，不能访问实例字段，因为静态方法可以在没有创建任何对象的情况下直接调用，而实例字段必须在对象创建之后才会分配内存空间，因而不能访问。

静态方法只能调用本类中的其他静态方法，不能调用实例方法，因为实例方法可能会间接访问实例字段。

4. 静态成员的应用

(1) 以类的形式管理全局变量或外部函数。

(2) 将具有相同属性值的字段提炼出来，定义成静态字段，这样可以让所有对象共用同一个静态字段，从而减少内存占用。

(3) 将专门处理静态字段的方法定义成静态方法。

例如，Java 语言自己就以静态成员的语法形式，将一些常用的数学函数和常量封装在一起，定义了一个数学类 Math。例 3-14 给出了一个使用数学类 Math 中静态成员的演示程序。

例 3-14　一个使用数学类 Math 中静态成员的演示程序

```
 1  public class MathTest {                     //测试类：测试 Java 语言中数学类 Math 的静态成员
 2
 3      public static void main(String[] args) {                    //主方法
 4          System.out.println( "random() = " + Math.random() ); //随机数函数(静态方法)
 5          System.out.println( "random() = " + Math.random() );
 6
 7          System.out.println( "sqrt(36) = " + Math.sqrt(36) ); //求平方根函数(静态方法)
 8          //下面的语句中使用了正弦函数(静态方法)和常量 PI(静态字段)
 9          System.out.println( "sin(30) = " + Math.sin(30 * Math.PI/180) );
10      }
11  }
```

本节习题

1. 下列关于类定义语法的描述中，错误的是(　　)。

   A. 定义类时需使用关键字 class

   B. 类成员包括字段成员和方法成员两种

   C. 类成员的访问权限有 4 种

   D. 类的访问权限有 4 种

2. 下列关于字段成员的描述中，错误的是(　　)。

   A. 字段相当于是类中的全局变量，用于保存数据

B. 字段不能与其他类成员重名

C. 定义字段的语法形式类似于定义变量,但定义时不能初始化

D. 未初始化的字段会被自动初始化成空值

3. 下列关于方法成员的描述中,错误的是(　　)。

A. 方法相当于是类中的函数,其功能通常是对字段成员进行处理

B. 方法包括 4 大要素,分别是方法名、形式参数列表、方法体和返回值类型

C. 方法可直接访问本类中的任意字段,访问时不受权限约束

D. 方法成员不能与类中的其他方法成员重名

4. 下列关于对象的描述中,错误的是(　　)。

A. 对象是用类定义的变量,也可称为是类的实例

B. 一个对象只属于某一个类

C. 一个类只能定义一个对象

D. 新建对象必须使用运算符 new 来为对象动态分配内存

5. 下列关于对象的描述中,错误的是(　　)。

A. 对象包含哪些成员是由其类定义决定的

B. 对象名实际上是对象的引用变量名

C. 对象的方法成员用于处理数据,通过"对象名.方法成员名()"进行调用

D. 可以调用对象中的所有方法成员

6. 下列关于 Java 语言数据类型的描述中,错误的是(　　)。

A. Java 语言中的数据类型分为基本数据类型和引用数据类型两大类

B. 定义基本数据类型的变量时直接为变量分配内存单元

C. 定义引用数据类型的对象时必须使用运算符 new 来动态分配内存单元

D. 引用变量保存某个对象的引用,引用变量不单独占用内存单元

7. 下列关于对象引用的描述中,错误的是(　　)。

A. 运算符 new 在创建对象后将返回该对象的引用

B. 一个对象可以被多个引用变量同时引用

C. 引用变量在引用一个对象之后不能再改变引用,引用其他对象

D. 当一个对象不被任何变量引用时,其内存单元将被 Java 虚拟机收回

8. 假设新建一个类 Circle 的对象 obj,下列写法中错误的是(　　)。

A. Circle obj = new Circle();

B. Circle obj; obj = new Circle();

C. Circle obj = new Circle;

D. Circle obj = new Circle(), obj1 = obj;

9. 下列关于 Java 语言中变量的描述中,错误的是(　　)。

A. Java 语言中的变量分为字段、局部变量和形参 3 种

B. 字段可以是基本数据类型,也可以是引用数据类型

C. 局部变量可以是基本数据类型,也可以是引用数据类型

D. 形参只能是基本数据类型,不能是引用数据类型

10. 下列关于参数传递的描述中,错误的是(　　)。

A. Java 语言中方法间传递基本数据类型数据时直接传递数值，即值传递

B. Java 语言中方法间传递引用数据类型数据时传递的是对象引用，即引用传递

C. 引用传递后，形参和实参将引用不同的对象

D. Java 语言中，当返回值是引用数据类型时返回的是对象引用

11. 下列关于对象内存分配的描述中，错误的是(　　)。

A. 新建对象时，Java 虚拟机会为对象的字段成员分配内存

B. 新建对象时，Java 虚拟机会为对象的方法成员分配内存

C. 内存中，同类的多个对象都包含各自的字段成员

D. 同类的多个对象在内存中会共用一份方法代码

12. 下列关于构造方法的描述中，错误的是(　　)。

A. 构造方法的名字必须与类名相同

B. 构造方法通过形参传递初始值，实现对新建对象字段成员的初始化

C. 构造方法没有返回值，其返回值类型应当写 void

D. 构造方法可以重载，这样可以提供多种形式的初始化方法

13. 假设类 Circle 只定义了一个“Circle(int x) { … }”形式的构造方法，则下列新建对象语句中错误的是(　　)。

A. Circle obj = new Circle(10);　　B. Circle obj; obj = new Circle(10);

C. Circle obj = new Circle();　　D. Circle obj = new Circle(10/3);

14. 下列关于静态成员的描述中，错误的是(　　)。

A. Java 语言是纯面向对象的语言，程序中没有游离在类外的全局变量和外部函数

B. 在需要用到全局变量或外部函数的场合，可以将它们定义成类的静态成员

C. 在类中定义静态成员时，需使用关键字 public 进行限定

D. 静态成员是被本类所有对象共用的成员

15. 下列关于访问静态成员的描述中，错误的是(　　)。

A. 在本类方法成员中访问静态成员直接使用成员名访问，访问时不受权限约束

B. 在类外其他方法中访问静态成员可以通过类名进行访问

C. 在类外其他方法中访问静态成员可以通过任何一个该类对象引用进行访问

D. 在类外其他方法中访问静态成员，访问时不受权限约束

3.4 数组

数组(array)是一组类型相同并按某种次序排列的数据集合，其中的每个数据被称为数组的一个**元素**(element)。数组元素按排列次序编号。编号为从 0 开始的整数，被称作数组元素的**下标**(subscript)。

存储一维方向排列的数据(例如数列)使用**一维数组**，一维数组有 1 个下标；存储二维方向排列的数据(例如矩阵)使用**二维数组**，二维数组有 2 个下标，第 1 个为行下标，第 2 个为列下标，……。Java 语言可以定义多维数组，但最常用的是一维数组和二维数组。

Java 程序中可以定义**数组变量**(简称为**数组**)来保存数据集合。对数据集合最常规的处理方法是依次访问集合中的每个元素，将所有元素逐个处理一遍，这种处理方法称为对数

据集合的**遍历**。

3.4.1 定义数组

1. 定义数组的步骤

Java 语言中，数组属于引用数据类型。定义数组需分两步完成。

(1) 先定义数组类型的引用变量。

定义一个数组类型的引用变量，定义时指定数据类型，并在引用变量名后面(或前面)加一对空的中括号“[ ]”。数组类型的引用变量名也被称作**数组名**，需符合标识符的命名规则。例如：

```
int iArray[ ];                    //定义一个 int 型数组的引用变量 iArray
```

或

```
int [ ]iArray;                    //定义一个 int 型数组的引用变量 iArray
```

(2) 再创建数组。

定义好数组类型的引用变量之后，再使用运算符 new 创建数组，为数组动态分配内存。创建数组时需指定数据类型和元素个数。例如：

```
iArray = new int[ 5];
```

计算机执行该语句，将创建一个包含 5 个元素的 int 型数组，并将运算符 new 返回的数组引用赋值给引用变量 iArray。这时可以说 iArray 是一个包含 5 个元素的 int 型数组。

可以将“定义引用变量”和“创建数组”这两步合并成一步完成。例如：

```
int iArray[ ] = new int[ 5];      //定义引用变量的同时创建数组
```

注意：定义引用变量和创建数组时所使用的数据类型应当一致。

2. 定义数组时的初始化

定义数组时可以初始化。例如：

```
int iArray[ ] = { 2, 4, 6 };
```

使用大括号“{ }”给出各数组元素的初始值，初始值之间用逗号“,”隔开。

给定初始值时，编译器将自动创建数组(不需要使用运算符 new)，数组的大小等于初始值的个数。换句话说，如果定义数组时初始化，程序员不需要使用运算符 new，也无法指定元素个数。数组的元素个数由初始值的数量决定。

3. 数组的语法细则

(1) 数组及其引用变量各自分配不同的内存单元。

(2) 数组属于引用数据类型，多个引用变量可以引用同一个数组对象实例。例如：

```
int iArray[ ] = new int[5];      //定义引用变量 iArray,并引用一个新建的数组对象一
int aRef[ ] = iArray;            //再定义一个引用变量 aRef,同时引用数组对象一
```

iArray 和 aRef 引用同一个数组对象，数组对象一的引用计数为 2(见图 3-11)。

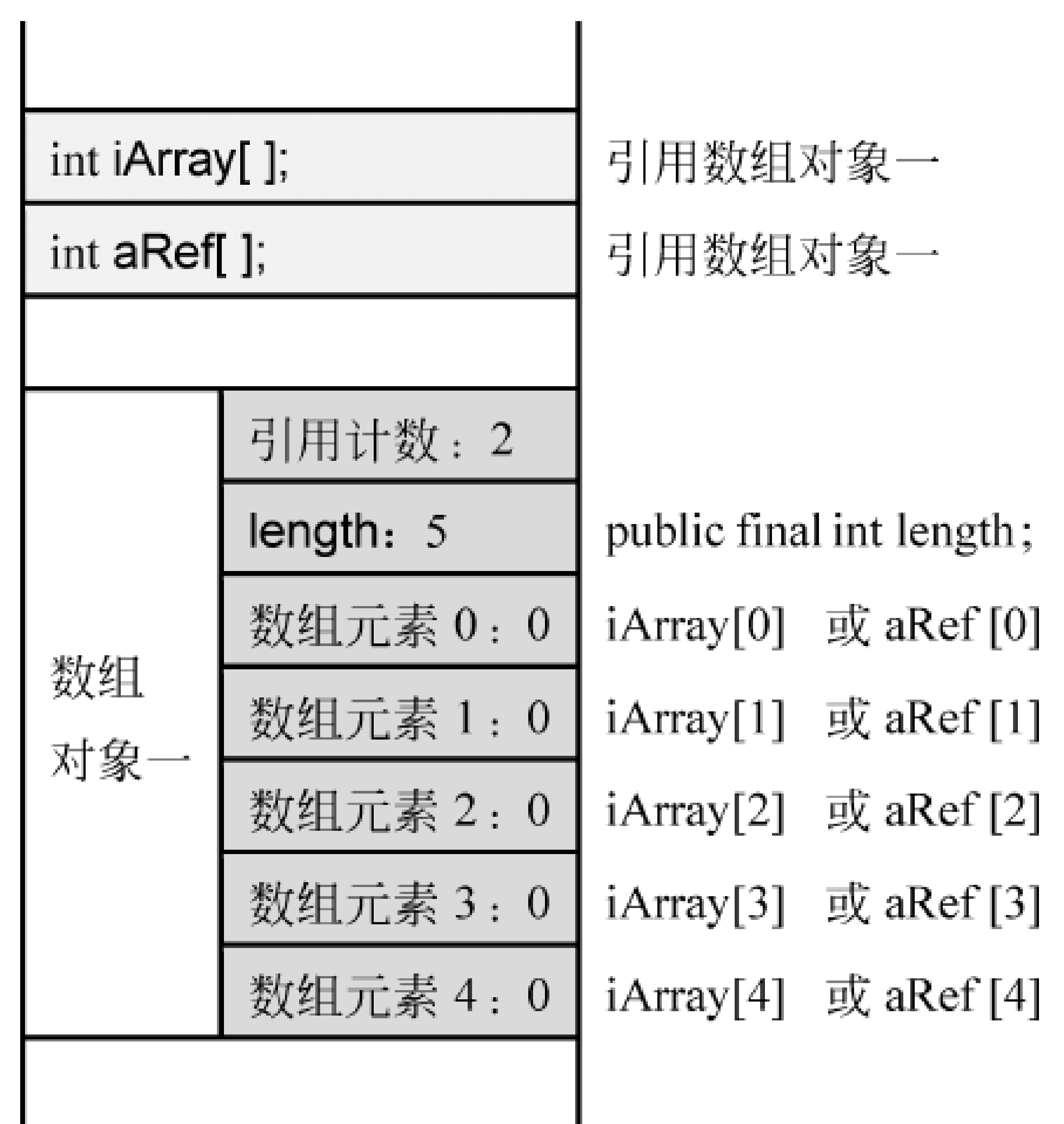


图 3-11　多个引用变量可引用同一数组

(3) 在 Java 语言内部，数组实际上是一种类类型，其中包含一个字段成员 **length**(只读字段)。字段 length 中自动存放了数组元素的个数，这个元素个数被称为数组的长度(或称为数组的大小)。例如，可以按如下形式显示图 3-11 中数组元素的个数：

```
System.out.println( iArray.length );     //显示 5
System.out.println( aRef.length );       //显示 5
```

(4) 数组中的元素按下标顺序在内存中连续存放。下标从 0 开始，到 length－1(即元素个数减 1)结束。

(5) 定义数组保存一组数据，要求每个数据的类型必须相同。数组只能保存具有相同数据类型的数据集合。

(6) 使用运算符 new 创建数组时，各数组元素会被自动初始化为**空值**(见表 3-1)。

3.4.2　访问数组

1. 访问数组中的元素

数组中保存的是一个数据集合，各数组元素按下标顺序连续排列。可以通过下标来指定访问某个数组元素。访问时，将下标用一对中括号“[ ]”括起来。访问包括写入数据或读出数据。例如：

```
int iArray[ ] = new int[ 5];             //定义一个 int 数组 iArray,其中包含 5 个元素
```

数组 iArray 中的 5 个元素依次为：

```
iArray[0]、iArray[1]、iArray[2]、iArray[3]、iArray[4]
```

创建数组 iArray 时，其中的各数组元素都被自动初始化为空值(int 型的空值是 0)。

```
iArray[ 0] = 10 ;                    //写入数据:向第 0 个元素的内存单元写入数据 10
iArray[ 1] = 20 ;                    //写入数据:向第 1 个元素的内存单元写入数据 20
System.out.println( iArray[ 0] );    //读出数据并显示,显示结果 10
System.out.println( iArray[ 1] );    //读出数据并显示,显示结果 20
System.out.println( iArray[ 2] );    //读出数据并显示,显示结果 0
```

数组 iArray 包含 5 个元素,其下标范围是 0~4。访问数组元素时,下标不能超出这个范围,即下标不能**越界**。例如:

```
System.out.println( iArray[ 5 ] );   //错误:下标越过了上界 4
System.out.println( iArray[ -1 ] );  //错误:下标越过了下界 0
```

编写程序时,数组下标越界是一种严重错误。编译器不能检查出下标越界错误,程序执行时将会中途出错退出。

数组 iArray 可以被多次引用。例如:

```
int aRef[ ] = iArray;                //再定义一个引用变量 aRef,同时引用数组 iArray
System.out.println( aRef[0] );       //读出 aRef[0](即 iArray[0])并显示,显示结果 10
```

在这个例子中,aRef 和 iArray 引用了同一个数组对象,因此访问效果是一样的。

2. 数组的遍历

很多情况下,程序需要遍历数组中保存的数据集合,即依次访问数组中的元素,将所有元素逐个处理一遍。设计遍历算法需要用到循环结构。例 3-15 给出一个遍历数组的 Java 演示程序。

例 3-15 一个遍历数组的 Java 演示程序(ArrayDemo.java)

```
1   public class ArrayDemo {                                   //主类
2       public static void main(String[ ] args) {              //主方法
3           int iArray[ ] = { 2, 4, 6, 8, 10 };    //定义一个 int 型数组 iArray,定义时初始化
4           for (int n = 0; n < iArray.length; n++)            //遍历数组,逐个显示各数组元素的值
5               System.out.println( iArray[ n] );              //显示第 n 个元素的值
6           //遍历数组,计算各数组元素的累加和
7           int sum = 0;
8           for (int n = 0; n < iArray.length; n++)
9               sum += iArray[ n];                             //累加第 n 个数组元素
10          System.out.println( sum );
11
12          char cArray[ ] = { 'C', 'h', 'i', 'n', 'a' };     //定义一个字符型数组 cArray
13          for (int n = 0; n < cArray.length; n++) {          //遍历数组,将所有小写字母改为大写
14              if (cArray[ n] >= 'a' && cArray[ n] <= 'z')    //检查是否小写字母
15                  cArray[ n] -= 32;                          //如果是,则将小写字母改为大写
16          }
17          System.out.println( cArray );          //只有字符数组才能整体输出,显示结果: CHINA
18  } }
```

3. 增强 for 语句

针对数据集合遍历算法，Java 语言还提供了一种增强的 for 语句。将例 3-15 的 for 语句改成增强 for 语句，程序功能保持不变。例 3-16 给出改写后的程序代码。

例 3-16 一个遍历数组(增强 for 语句)的 Java 演示程序(EnhancedFor.java)

```
1  public class EnhancedFor {                         //主类
2      public static void main(String[ ] args) {  //主方法
3          int iArray[ ] = { 2, 4, 6, 8, 10 };  //定义一个 int 型数组 iArray,定义时初始化
4          for (int x: iArray)    //增强 for 语句:依次将数组 iArray 中的元素取出来,赋值给 x
5              System.out.println( x );          //显示所取出的值
6          //遍历数组,计算各数组元素的累加和
7          int sum = 0;
8          for (int x: iArray)                        //增强 for 语句:计算各数组元素的累加和
9              sum += x;                              //累加所取出的值
10         System.out.println( sum );
11
12         char cArray[ ] = { 'C', 'h', 'i', 'n', 'a' };            //定义一个字符型数组 cArray
13         for (char x: cArray) {//增强 for 语句:依次将数组 cArray 中的元素取出来,赋值给 x
14             if ( x>= 'a' && x<= 'z')           //检查所取出的字符是否是小写字母
15                 x -= 32;    //将小写字母改为大写. 注:此处只能修改 x,无法修改数组元素
16         }
17         System.out.println( cArray );          //显示结果: China
18         //可以看出,当需要修改数组元素时,还是只能用普通 for 语句
19 } }
```

3.4.3 可变长形参

通常，一个方法中的形参个数是确定的。例如：

```
int max(int x, int y)  { … }             //求 2 个数最大值的方法:有 2 个形参
int max(int x, int y, int z)  { … }      //求 3 个数最大值的方法:有 3 个形参
```

是否可以定义一个求任意多个数最大值的方法呢？例如：

```
int max(int x1, int x2, … )  { … }       //求任意多个数最大值的方法:形参个数不确定
```

在这个求最大值的方法中，形参的个数是不确定的，称这样的形参是**可变长形参**(variable arguments)。Java 语言提供了一种定义可变长形参的语法形式。

```
int max(int … varArgs) {                 //定义可变长形参,在形参名前加"…"(3 个点)
    …                                    //求最大值算法
}
```

例 3-17 给出了一个完整的具有可变长形参的求最大值方法 Java 演示程序。

例 3-17 一个具有可变长形参的求最大值方法 Java 演示程序(VarArgument.java)

```
1  public class VarArgument {                              //主类
2      public static int max(int...varArgs) {              //具有可变长形参的求最大值方法
3          //可变长形参 varArgs 所接收到的实参是以数组形式存放的,varArgs 是一个数组
4          if (varArgs.length < 1)   return 0;             //如果没有传递实参,则直接返回 0
5          int result = varArgs[0];                        //先假设第 0 个元素就是最大值
6          for (int n = 1; n < varArgs.length; n++) {      //求数组元素中的最大值
7              if (varArgs[ n] > result)   result = varArgs[ n];
8          }
9          /* 也可使用以下的增强 for 语句来求最大值
10         for (int e : varArgs)
11         {   if ( e > result)   result = e;   }
12         */
13         return result;
14     }
15
16     public static void main(String[] args) {            //主方法
17         System.out.println( max(2, 4) );                //传递 2 个实参,显示结果 4
18         System.out.println( max(2, 4, 6) );             //传递 3 个实参,显示结果 6
19         System.out.println( max(2, 4, 6, 8) );          //传递 4 个实参,显示结果 8
20         System.out.println( max(2) );                   //传递 1 个实参,显示结果 2
21         System.out.println( max() );                    //不传递实参,显示结果 0
22     }
23 }
```

可变长形参的语法细则如下。

(1) 可变长形参中,形参的个数可变,但要求各形参的数据类型是相同的,即只能有一种数据类型。

(2) 调用可变长形参的方法时,可以向可变长形参传递 0 个或任意多个实参。

(3) 可变长形参以数组的形式来接收实参,即可变长形参是一个数组。该数组按位置顺序依次保存调用方法时所传递的实参值,通过数组的属性成员 **length** 可以得到实参的个数。

3.4.4 二维数组

存储二维表格、矩阵这样的数据集合需要使用二维数组。二维数组有两个下标,第一个为**行**下标,第二个为**列**下标。

1. 定义二维数组

Java 语言用两对中括号"[ ]"来表示二维数组。创建数组时需分别指定数组的行数和列数,先指定行数,再指定列数。二维数组的元素个数=行数×列数。例如:

```
int iArray[ ][ ];                    //定义一个 int 型二维数组的引用变量 iArray
iArray = new int[ 2][ 3];            //创建一个 2 行 3 列的 int 型二维数组
```

可以将这两条语句合并成如下的一条语句:

```
int iArray[ ][ ] = new int[2][3];        //定义引用变量的同时创建数组
```

计算机执行该语句，将创建一个2行3列的int型二维数组，并将运算符new返回的数组引用赋值给引用变量iArray。这时可以说：iArray是一个2行3列的int型二维数组。

定义二维数组时可以初始化。例如：

```
int iArray[ ][ ] = { { 1, 3, 5 }, { 2, 4, 6 } };     //定义二维数组时给出初始值
```

使用大括号"{ }"并按行的顺序给出每一行元素的初始值，初始值之间用逗号","隔开。如果给定了初始值，编译器将自动创建数组(不需要使用运算符new)，二维数组的大小等于初始值列出的行数和列数。

2. 理解二维数组

(1) 二维数组的每一行都可以看作是一个一维数组。例如：

```
int a[ ][ ] = new int[ 2][ 3];               //定义一个2行3列的二维数组a
```

等价于

```
int a[ ][ ] = new int[ 2][ ];                //先定义一个2行的二维数组
a[ 0] = new int[ 3];                         //再定义第0行的一维数组,包含3个元素
a[ 1] = new int[ 3];                         //再定义第1行的一维数组,包含3个元素
System.out.println( a.length );              //显示数组a的行数2
System.out.println( a[0].length );           //显示第0行的列数3
System.out.println( a[1].length );           //显示第1行的列数3
```

(2) 二维数组每一行的列数可以不同。例如：

```
int a[ ][ ] = new int[ 2][ ];                //先定义一个2行的二维数组
a[ 0] = new int[ 3];                         //再定义第0行的一维数组,包含3个元素
a[ 1] = new int[ 5];                         //再定义第1行的一维数组,包含5个元素
System.out.println( a.length );              //显示数组a的行数2
System.out.println( a[0].length );           //显示第0行的列数3
System.out.println( a[1].length );           //显示第1行的列数5
```

3. 访问二维数组元素

访问二维数组中的元素需指定行下标和列下标，其访问形式为：

数组名[行下标][列下标]

例如，访问前面定义的二维数组iArray，其中包含2行3列，共6个元素。访问这些数组元素的形式如下。

第0行的3个元素：iArray[**0**][0]、iArray[**0**][1]、iArray[**0**][2]。

第1行的3个元素：iArray[**1**][0]、iArray[**1**][1]、iArray[**1**][2]。

例3-18给出二维数组的Java演示程序。遍历二维数组需使用两重循环。

例 3-18 一个二维数组的 Java 演示程序(Array2D.java)

```
1  public class Array2D {                                        //主类
2      public static void main(String[ ] args) {                 //主方法
3          //定义一个 2 行 3 列的二维数组 a(可认为是一个矩阵),定义时初始化
4          int a[ ][ ] = { { 1, 3, 5 }, { 2, 4, 6 } };
5          //数组遍历:按先行后列的次序显示各数组元素的值,需使用两重循环
6          for (int m = 0; m < a.length; m++) {                  //行循环:m 保存行下标
7              for (int n = 0; n < a[m].length; n++)             //列循环:n 保存列下标
8                  System.out.print( a[m][n] + " " );
9              System.out.println();                             //换行显示下一行数组元素
10         }
11         //将二维数组(矩阵)a 转置后保存到 b 中
12         int b[ ][ ] = new int[3][2];                           //定义 3 行 2 列的二维数组 b
13         //数组遍历:计算转置矩阵 b
14         for (int m = 0; m < b.length; m++) {                  //行循环:m 保存行下标
15             for (int n = 0; n < b[m].length; n++) {           //列循环:n 保存列下标
16                 b[m][n] = a[n][m];   //将 a 和 b 的行列下标互换后赋值,这就是矩阵的转置
17                 System.out.print( b[m][n] + " " );
18             }
19             System.out.println();                             //换行显示下一行数组元素
20         }
21 } }
```

3.4.5 对象数组

可以定义类类型的数组,这就是对象数组。

1. 定义对象数组

用类所定义出的数组就是对象数组,其定义语法与基本数据类型数组基本一样。例如:

```
Clock c[ ];                    //定义一个 Clock 类型对象数组的引用变量 c
c = new Clock[3];              //创建一个包含 3 个元素的 Clock 类型对象数组
```

可以将这两条语句合并成如下一条语句:

```
Clock c[ ] = new Clock[3];     //定义引用变量的同时创建对象数组
```

计算机执行该语句,将创建一个包含 3 个元素的 Clock 类型数组,并将运算符 new 返回的数组引用赋值给引用变量 c。这时可以说 c 是一个包含 3 个元素的 Clock 类型数组。

与基本数据类型不同的是,对象数组中的元素还只是引用变量,具体的对象仍需要继续使用运算符 new 单独创建。例如:

```
c[0] = new Clock();            //创建第一个 Clock 对象,并将其引用赋值给 c[0]
c[1] = new Clock( 8, 30, 15 ); //创建第二个 Clock 对象,并将其引用赋值给 c[1]
c[2] = new Clock( 12, 0, 0 );  //创建第三个 Clock 对象,并将其引用赋值给 c[2]
```

到这里才最终完成对象数组 c 的定义和创建工作。对象数组 c 的内存分配示意图如图 3-12 所示。

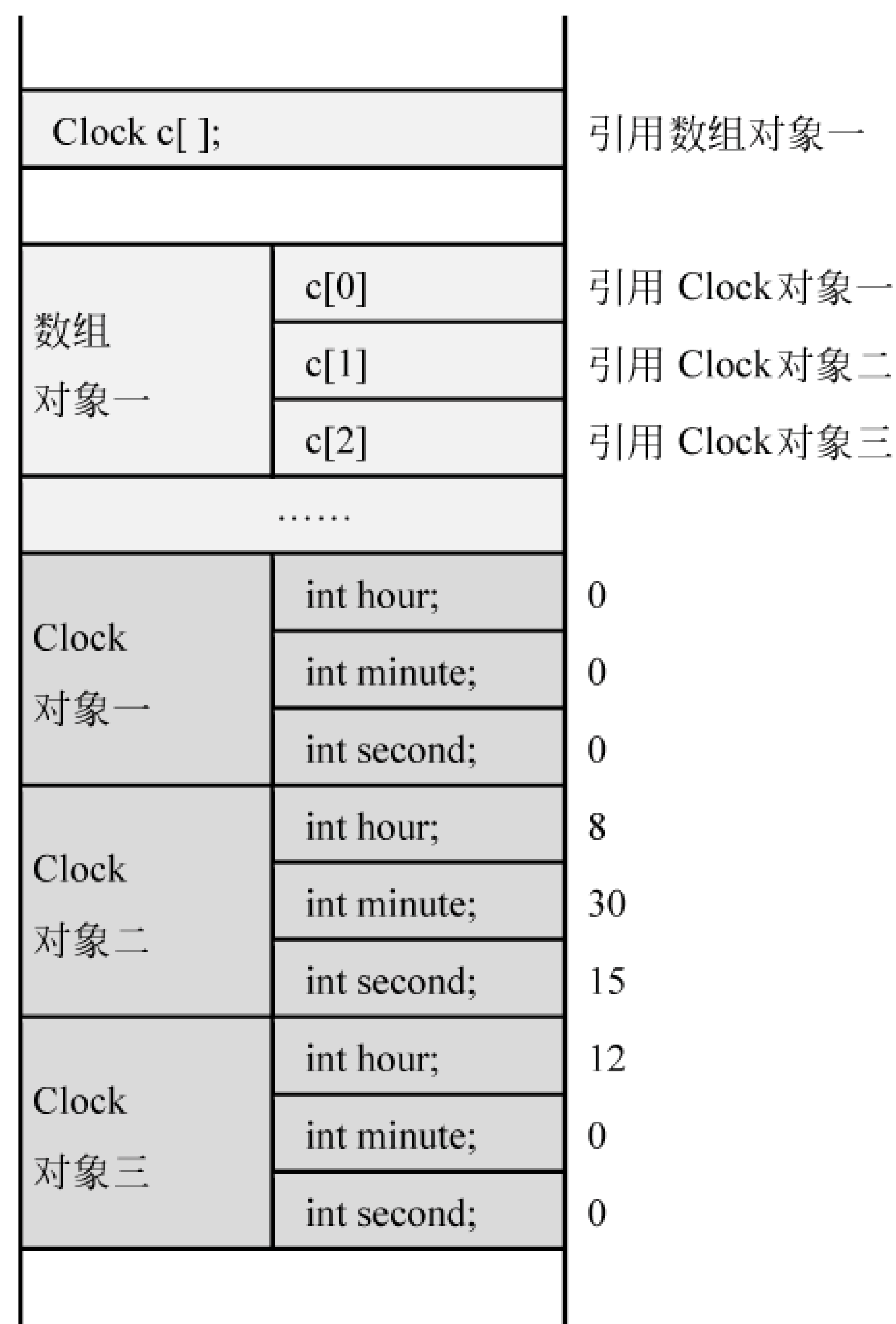


图 3-12　对象数组 c 的内存分配示意图

定义对象数组时可以通过**初始化**直接创建对象。例如，使用如下形式的一条语句就可以完成上述对象数组 c 的定义和创建工作。

```
Clock c[ ] = { new Clock(), new Clock( 8, 30, 15 ), new Clock( 12, 0, 0 ) };
```

2. 访问对象数组

访问对象数组通常分为如下两个层次。

（1）访问数组元素。对象数组中的每个元素都是一个对象引用，其访问形式是：

数组名[下标]

（2）访问数组元素的下级成员。数组元素是对象引用，可以进一步访问数组元素所引用对象的下级成员，例如访问对象的字段或调用对象的方法。其访问形式是：

数组名[下标].字段名
数组名[下标].方法名(…)

例 3-19 给出一个对象数组的 Java 演示程序。

例 3-19 一个对象数组的 Java 演示程序(ArrayObject.java)

```
1  public class ArrayObject {                  //主类
2      public static void main(String[] args) {  //主方法
3          Clock c[ ] = new Clock[6];          //定义一个包含 6 个元素的钟表对象数组
4          //遍历数组:创建钟表对象,设置并显示其时间。各钟表对象的时差为 1 小时
5          for (int n = 0; n < c.length; n++) {
6              c[n] = new Clock();            //创建钟表对象,将其引用赋值给第 n 个元素
7              c[n].set(n, 0, 0);             //设置钟表对象的时间
8              c[n].show();                   //显示钟表对象的时间
9          }
10 } }
```

本节习题

1. 定义一个包含 3 个元素的 char 型数组 x,下列写法中正确的是(　　)。
   A. char x = new char[3];　　B. char x[3]; x = new char[];
   C. char x = new char(3);　　D. char x[]; x = new char[3];
2. 定义一个包含 3 个元素的 double 型数组 x,下列写法中正确的是(　　)。
   A. double x[] = {1.5, 2.5, 3.5};
   B. double x[3];　x = {1.5, 2.5, 3.5};
   C. double x[3] = new {1.5, 2.5, 3.5};
   D. double x[3];
3. 定义一个包含 3 个元素的 double 型数组 x,下列访问数组元素的形式中错误的是(　　)。
   A. x[0]　　B. x[1]　　C. x[2]　　D. x[3]
4. 定义一个具有 int 型可变长形参的方法 fun(),下列写法中正确的是(　　)。
   A. int fun(int x1, int x2, …) { … }　　B. void fun(int …x[]) { … }
   C. int …fun(int x[]) { … }　　D. void fun(int …x) { … }
5. 定义一个 2 行 3 列的 int 型数组 x,下列写法中错误的是(　　)。
   A. int x[][] = new int[2][3];
   B. int x[][] = new int[3][2];
   C. int [][]x = new int[2][3];
   D. int x[][] = {{1, 2, 3}, {4, 5, 6}};
6. 如果想获取一个 2 行 3 列数组 x 的行数,下列写法中正确的是(　　)。
   A. x.length　　B. x[0].length　　C. x[1].length　　D. x.length()
7. 下列关于对象数组的描述中,错误的是(　　)。
   A. 对象数组中的每个元素都是一个对象引用
   B. 对象数组中的每个元素都是一个对象
   C. 可以访问对象数组中的数组元素
   D. 可以访问对象数组中数组元素的下级成员

8. 定义一个包含 3 个元素的类 Circle 的对象数组 x，下列写法中错误的是(　　)。

A. Circle x[]=new Circle[3];

B. Circle []x=new Circle[3];

C. Circle x[3]=new Cirlce();

D. Circle x[]={new Circle(), new Circle(), new Circle()};

3.5 Java 程序文件的组织

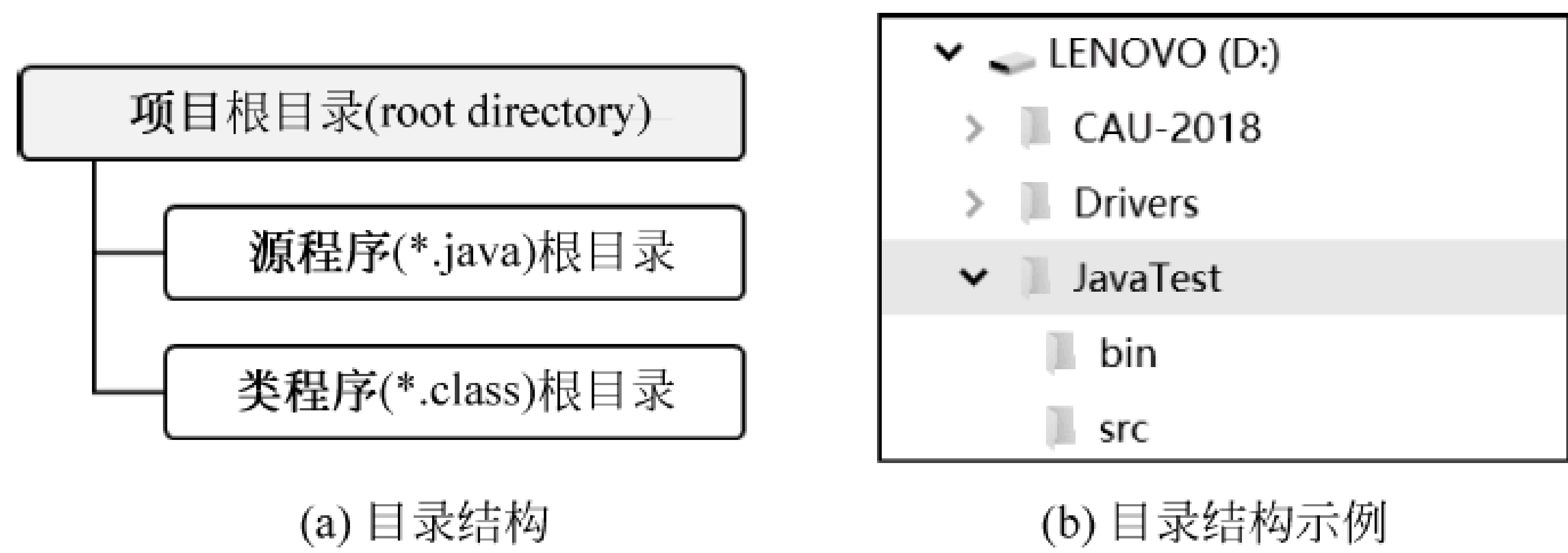
开发一个大型 Java 程序**项目**(project)可能要编写很多个**源程序文件**(source file)，这就是多文件结构的 Java 程序。源程序文件的扩展名为.java。

一个 Java 源程序文件可以包含多个**类**(class)，但其中最多只能有一个 public 类，此时源程序文件的文件名必须与这个 public 类的类名相同。

编译后，Java 源程序文件中的每个类都会生成(或称为输出，output)一个与类同名的**类程序文件**(class file)，其扩展名为.class。类程序文件所保存的是类编译之后的字节码指令。

3.5.1 Java 项目的目录结构

通常，将同一 Java 项目的程序文件放在一个**目录**(directory)下进行集中管理。目录也被称为**文件夹**(folder)。图 3-13 给出了一个 Java 项目常用的目录结构示意图。



(a) 目录结构　　(b) 目录结构示例

图 3-13　Java 项目常用的目录结构示意图

图 3-13(a)在项目根目录中又建立了两个子目录，并将它们分别作为存放**源程序**文件(*.java)和编译后**类程序**文件(*.class)的根目录。图 3-13(b)给出一个示例，其中的项目根目录为 D:\JavaTest，源程序根目录为 src，类程序根目录为 bin。

在 Eclipse 集成开发环境中新建 Java 项目，需分别指定项目根目录、源程序根目录和类程序根目录。图 3-14、图 3-15 演示了在 Eclipse 中新建一个 Java 项目 Project1，然后按图 3-13(b)所示的目录结构将项目根目录设为 D:\JavaTest(见图 3-14)，将源程序根目录设为 src，类程序根目录设为 bin(见图 3-15)。

在将项目 Project1 的根目录设为 D:\JavaTest 之后，Eclipse 界面中的类程序根目录 Project1/bin 对应的是文件系统中的 D:\JavaTest\bin。同理，源程序根目录 Project1/src 对应的是文件系统中的 D:\JavaTest\src。

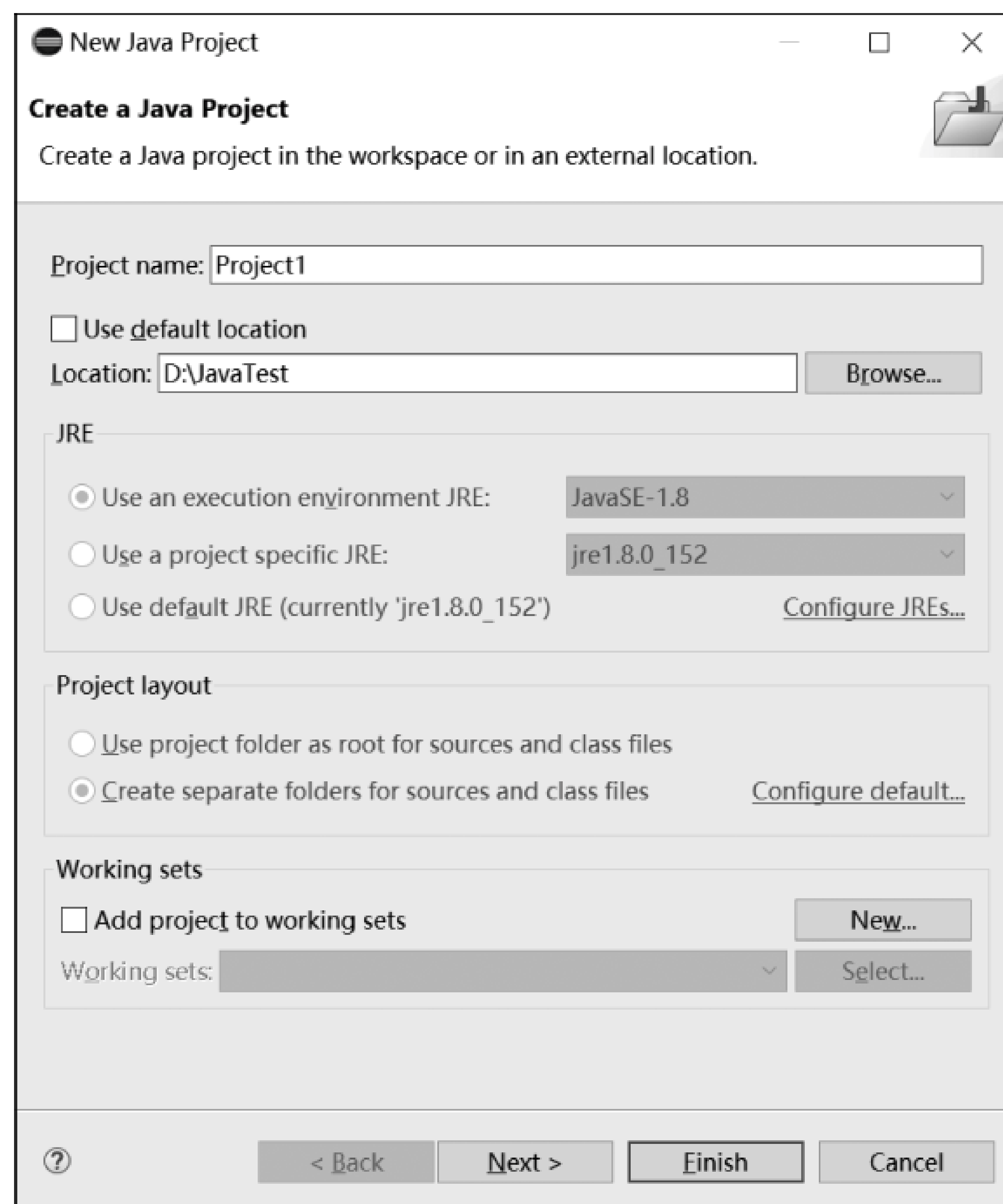


图 3-14　将项目 Project1 的根目录设为 D:\JavaTest

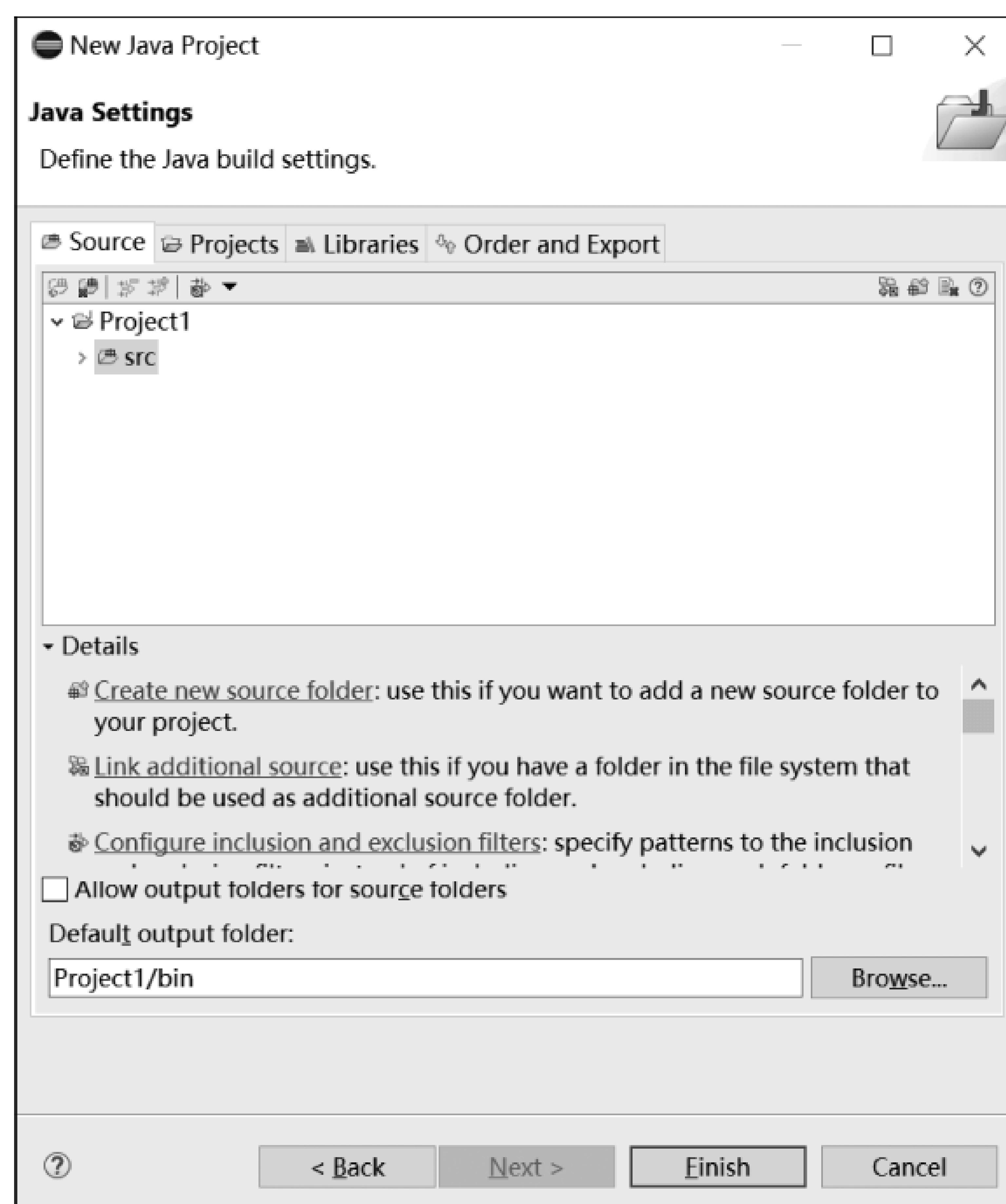


图 3-15　设置项目 Project1 的源程序根目录(src)和类程序根目录(bin)

3.5.2　在 Java 项目中添加 Java 类

在 Java 项目中添加(即新建)Java 类时,应当将类的源程序存放路径统一设为 Java 项目的源程序根目录,这样可以对项目的源程序文件进行集中管理。图 3-16 演示了在项目 Project1 中新建一个 Java 类 JClass1,并将该类的源程序文件存放文件夹(Source folder)设为 Project1/src。

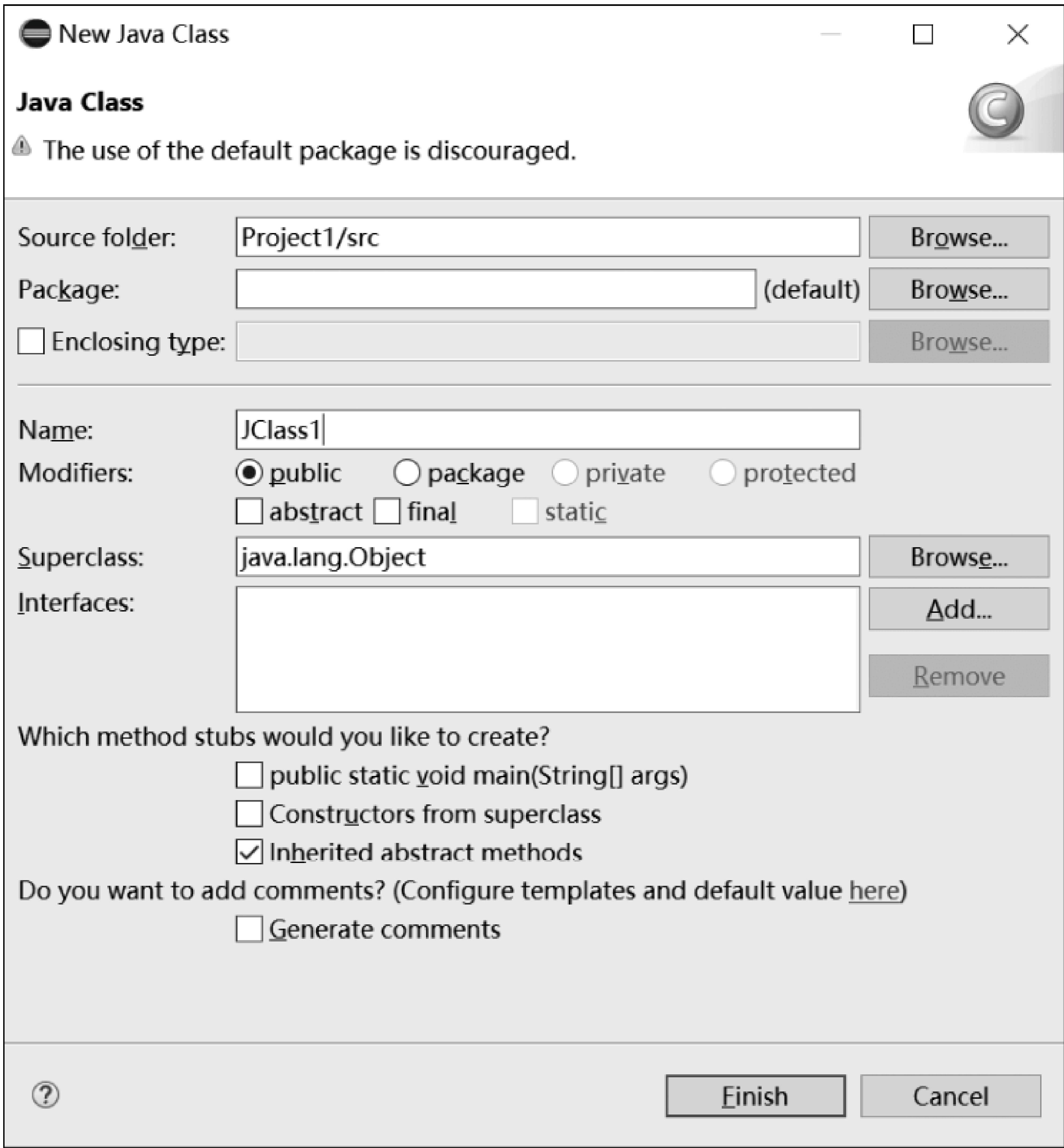


图 3-16　新建一个 Java 类 JClass1 并将其源程序文件保存到 Project1/src 中

假设,在项目 Project1 中添加两个类:JClass1 和 JMainClass,例 3-20 给出它们的完整定义代码。类 JMainClass 是一个主类,其中的主方法 main()用到了类 JClass1。

例 3-20　在项目 Project1 中添加两个类:JClass1 和 JMainClass

```
1  //类 JClass1:源程序文件 JClass1.java
2
3  public class JClass1 {
4      private int f1 = 10;      //一个字段
5      public void show1() {     //一个方法
6          System.out.println( "JClass1: " + f1 );
7      }
8  }
```

```
//主类 JMainClass:源程序文件 JMainClass.java
public class JMainClass {   //主类

    public static void main(String[ ] args) {
        JClass1 obj = new JClass1();
        obj.show1();
    }
}
```

在项目 Project1 中添加了类 JClass1、JMainClass 之后，查看项目 Project1 的源程序根目录 src(即 D:\JavaTest\src)，可以看到如图 3-17 所示的内容。



图 3-17 查看项目 Project1 源程序根目录 src 下的内容

运行主类 JMainClass，Eclipse 会先对 JMainClass. java 和 JClass1. java 进行编译，并将所生成的类程序文件自动保存到项目 Project1 的类程序根目录 bin(即 D:\JavaTest\bin)。查看该目录，可以看到如图 3-18 所示的内容。



图 3-18 查看项目 Project1 类程序根目录 bin 下的内容

Java 源程序文件的命名细则如下。

(1) 如果文件中有一个 public 类，则必须以该类的类名作为文件名。

(2) 如果文件中没有 public 类，则应任选文件中某个类的类名作为文件名。

(3) 一个 Java 源程序文件中最多只能有一个 public 类(即访问权限被定义为 public)，其他类都不能指定访问权限(即访问权限是默认权限)。

(4) Java 源程序文件的扩展名为. java。

(5) 编译后，Java 源程序文件中的每个类都会生成一个与类同名的类程序文件，扩展名为. class。

3.5.3 以包的形式管理 Java 类

大型 Java 程序会定义很多个类，会有很多个源程序文件。可以对这些源程序文件进行**分组**管理，即在**源程序根目录** src 下再建立**子目录**，将源程序文件分散到不同的子目录下进行管理。

1. 分包管理 Java 类

将 Java 源程序文件放入不同的子目录进行分组管理，实际上是对源程序文件中的类进行分组管理。将 Java 源程序文件放入不同的子目录，Java 语言称为“将文件中的类放入不同的**包**(package)”。源程序文件所在的**子目录名**被称为是类的**包名**。

例如，在项目 Project1(D:\JavaTest)的源程序根目录 src 下再建立一个子目录 lib1，然后在该子目录中新建两个类 JClass2 和 JClass3(见图 3-19)。这时，称“类 JClass2 和 JClass3 被放入了包 lib1 中”。



图 3-19　类 JClass2 和 JClass3 被放入了包 lib1 中

在 Eclipse 中将一个类放入某个包中，需要在新建 Java 类时为其指定包名。图 3-20 演示了新建 Java 类 JClass2 时将其放入在包 lib1 的设置界面，其中将与包相关的设置选项即 Package 选项设为 lib1。单击 Finish 按钮，Eclipse 将为类 JClass2 创建一个源程序文件 JClass2.java，并自动将该文件保存到源程序根目录 src 下的子目录 lib1 中。

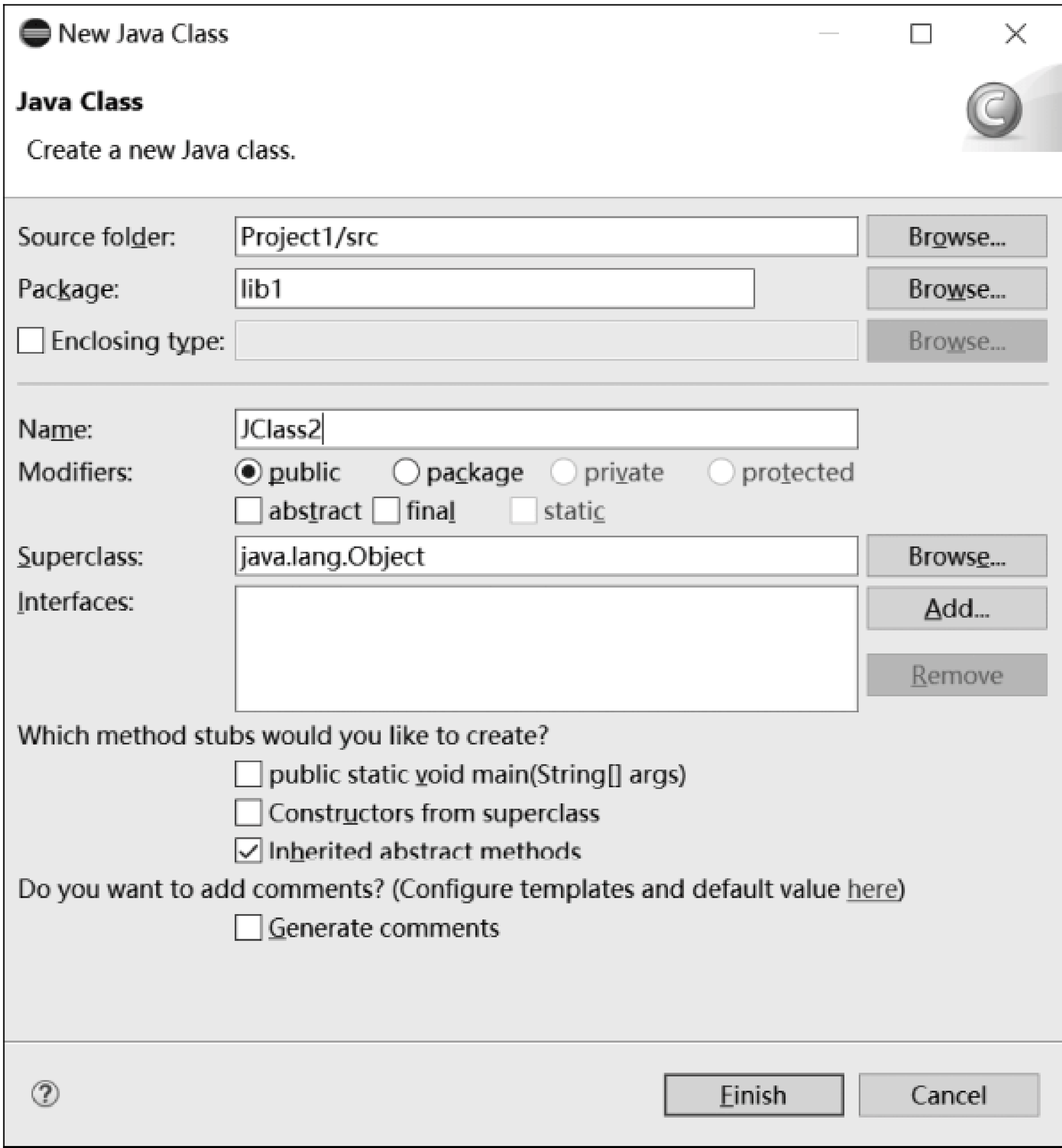


图 3-20　新建类 JClass2 时将 Package 选项设为 lib1

例 3-21 给出了类 JClass2 和 JClass3 的完整定义代码。

例 3-21 在项目 Project1 的包 lib1 中添加两个类：JClass2 和 JClass3

```
1  //类 JClass2:源程序文件 lib1\JClass2.java
2   package lib1;  //向编译器声明包名
3
4  public class JClass2 {
5       private int f2 = 20;      //一个字段
6       public void show2() {     //一个方法
7           System.out.println( "JClass2: " + f2 );
8       }
9  }
```

```
//类 JClass3:源程序文件 lib1\JClass3.java
package lib1;  //向编译器声明包名

public class JClass3 {
     private int f3 = 50;      //一个字段
     public void show3() {     //一个方法
       System.out.println( "JClass3: " + f3 );
     }
}
```

如果将类放在某个包中，则必须在其源程序文件的开头使用 package 语句声明包名。在 Eclipse 中新建 Java 类，如果在设置界面为 Package 选项设定了包名，则 Eclipse 会自动为类代码添加这条 package 语句。下面给出 package 语句的语法。

Java 语法：**package 语句**

```
package 包名;
```

语法说明：

- **package** 语句的作用是向编译器声明本文件中类所在的包名，它应当是源程序代码的第一条语句(注释除外)。package 是 Java 语言的关键字。
- **包名**指定了源程序文件(.java)所在的子目录名，该子目录名是在源程序根目录下的相对路径名。这时，称"源程序文件(.java)中的类被放入到了指定的包中"。
- 编译时，Java 编译器会按照 package 语句在类程序根目录下创建完全相同的子目录结构，并将编译生成的类程序文件(.class)自动放入对应的子目录中。
- 存放在源程序根目录下的类不需要添加 package 语句。Java 称"这些类被放入了**默认(default)**包(或称无名包)中"。
- 可以在源程序根目录下建立多级子目录。这时，包名的命名形式为"一级子目录.二级子目录……"子目录之间用点"."隔开。
- **包名**(同时也是类所在的子目录名)习惯上以小写字母开头，**类名**(同时也是类的程序文件名)习惯上以大写字母开头，这样可以很容易辨识出包名与类名。

编译例 3-21 的源程序文件 JClass2.java 和 JClass3.java，Java 编译器会按照 package 语句的指示在类程序根目录 bin 下创建一个子目录 lib1，并将编译生成的类程序文件(.class)自动放入这个子目录(见图 3-21)。

对比图 3-21 和图 3-19 可以发现，Java 项目中的类程序根目录 bin 会与源程序根目录 src 保持完全相同的目录结构。3.5.2 节曾经向项目 Project1 添加过两个类，即 JClass1 和 JMainClass，本节又添加了 JClass2 和 JClass3 这两个类。至此，项目 Project1 总共定义了 4 个类。图 3-22 给出了项目 Project1 的完整目录结构和文件列表。

图 3-22 中，JMainClass.java 和 JClass1.java 保存在源程序根目录 src 下，这表示类

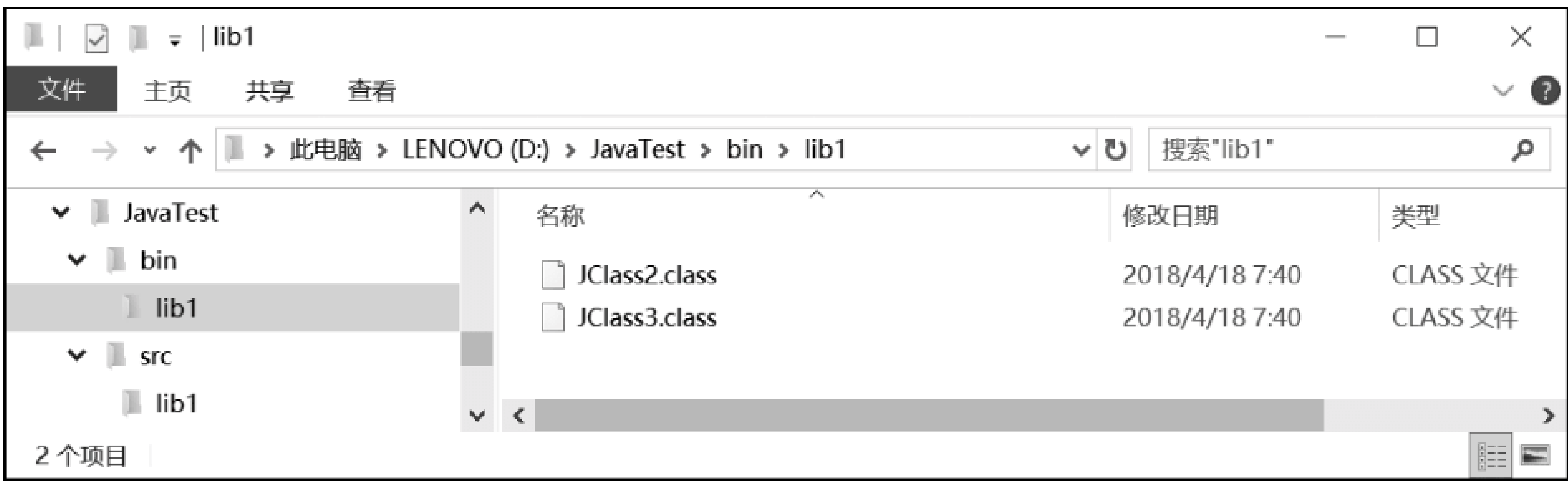


图 3-21　类程序根目录 bin 下的文件和目录结构

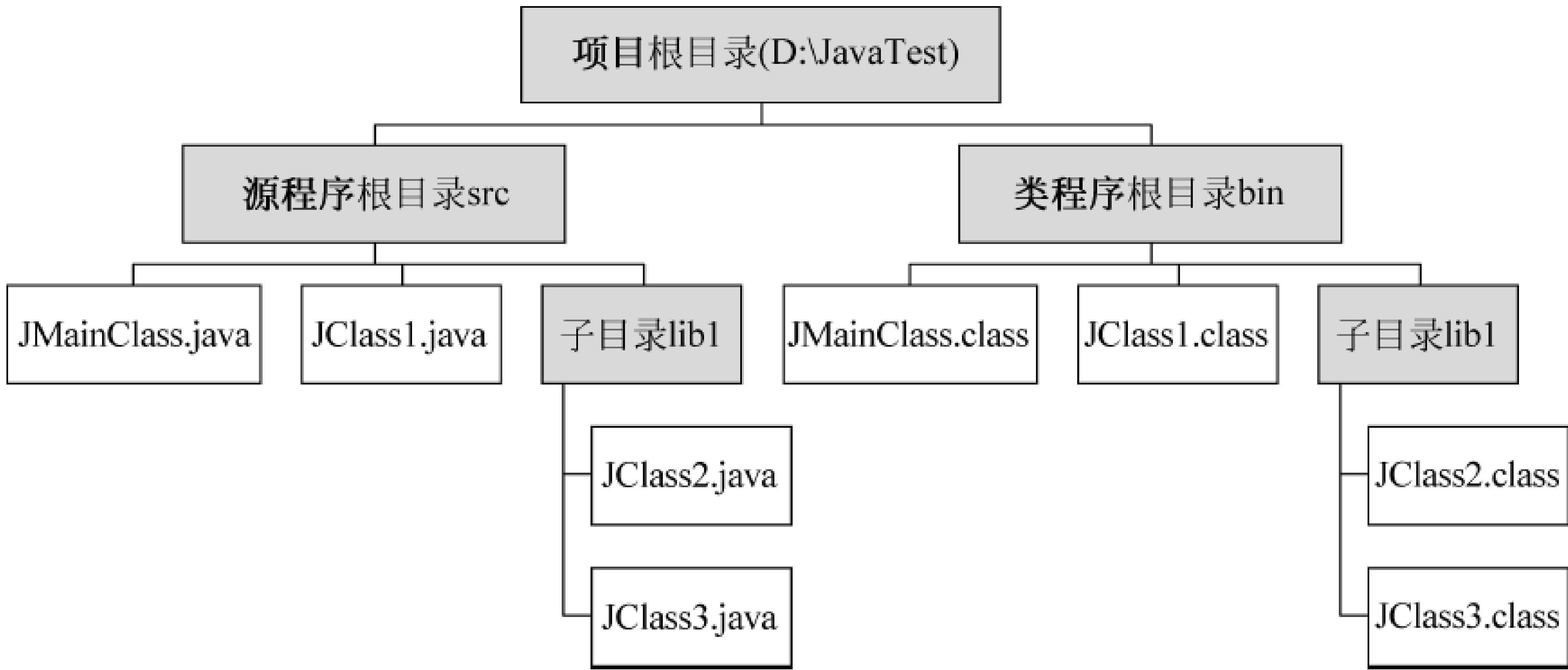


图 3-22　项目 Project1 的完整目录结构和文件列表

JMainClass 和 JClass1 都被放入了项目的默认包(或称无名包)。存放在默认包里的类不需要添加 package 语句。

而 JClass2.java 和 JClass3.java 保存在子目录 lib1 下,这表示类 JClass2 和 JClass3 被放入了 lib1 包。放在非默认包里的类必须在程序代码的开头添加 package 语句,声明包名。例如,类 JClass2 和 JClass3 都需要在程序代码的开头添加如下 package 语句:

```
package lib1;                    //将本文件中的类放入包 lib1
```

这里对分包管理 Java 类做一个小结:一个 Java 项目可能包含很多个 Java 类,可以对这些 Java 类进行分包管理,分包管理就是将其对应的程序文件放入不同的子目录;包名与目录名存在一一对应的关系;子目录下还可以再建立子目录,这相当于是在包里再划分子包,包可以分为任意多级,多级包名之间用"·"隔开;放入包中的类需要在其源程序文件的开头使用 package 语句声明其所在的包。

2. 使用不同包里的类

下面讨论如何使用不同包里的类。假设主类 JMainClass 要用到另外 3 个类 JClass1、JClass2 和 JClass3,其中 JClass1 与主类放在同一个包(即默认包)里。而 JClass2 和 JClass3 被放在包 lib1 里,与主类不在同一个包。使用不同包里的类,这与使用同一包里的类有什么区别吗?

1）使用同一包中的类时，直接使用类名

例如，主类 JMainClass 使用同一包里的类 JClass1 定义对象，可以直接使用类名。

```
JClass1 obj1 = new JClass1();
```

2）使用不同包里的类，需在类名之前加上包名

例如，主类 JMainClass 使用包 lib1 里的类 JClass2、JClass3 定义对象，需在类名之前加上包名，其语法形式为"**包名.类名**"。

```
lib1.JClass2 obj2 = new lib1.JClass2();
lib1.JClass3 obj3 = new lib1.JClass3();
```

3）使用不同包里的类可以预先导入，然后直接使用类名

Java 语言可以使用 import 语句预先导入不同包里的类，使用时省略包名，直接使用类名，这样可以简化程序代码。例 3-22 给出一个使用不同包里类的 Java 演示程序。

例 3-22 一个使用不同包里类的 Java 演示程序(JMainClass.java)

```
 1  import lib1.JClass2;                          //预先导入包 lib1 中的类 JClass2
 2  import lib1.JClass3;                          //预先导入包 lib1 中的类 JClass3
 3  //import lib1.*;                              //或者预先导入包 lib1 中的所有类
 4
 5  public class JMainClass {                     //主类
 6      public static void main(String[ ] args) { //主方法
 7          JClass1 obj1 = new JClass1();         //使用同一包里的类 JClass1,直接使用类名
 8          obj1.show1();
 9          //下面使用 lib1 包里的类 JClass2、JClass3
10          JClass2 obj2 = new JClass2(); //因为预先导入了 lib1 包里的类,这里可直接使用类名
11          JClass3 obj3 = new JClass3();
12          /* 如果不预先导入 lib1 包,则需在类名之前加上包名.例如
13          lib1.JClass2 obj2 = new lib1.JClass2();    //需在类名前加上包名 lib1
14          lib1.JClass3 obj3 = new lib1.JClass3();
15          */
16          obj2.show2();    obj3.show3();
17  } }
```

下面给出 import 语句的语法。

Java 语法：import **语句**

```
import 包名.类名;
import 包名.*;
```

语法说明：

- **import** 语句应放在源程序的开头(仅次于 package 语句)。import 是 Java 语言的关键字。
- "**包名.类名**"指定导入包中的某个类；"**包名.***"则是导入包中的所有类，但不包括其子包(即下级子目录)中的类。
- 后续程序使用被导入的类，可直接使用类名，这样可以简化程序代码；导入后仍可以继续使用"包名.类名"的形式。

■ 如果从不同包中导入了多个重名的类，此时必须使用"包名.类名"的形式，因为只有这样才能明确指定使用哪个包中的类。

注意：Java 语言中包的作用类似于 C++ 语言中的命名空间(namespace)，但 Java 里的包会实际对应文件系统中的目录(文件夹)，而 C++ 里的命名空间则不会。

3. 导入类中的静态成员

3.3.7 节例 3-14 曾给出一个使用数学类 Math 中静态成员的演示程序。访问其他类中的静态成员，需以"**类名.静态成员名**"的形式访问。例如：

```
System.out.println( Math.PI );          //访问 Math 中的静态字段 PI(π 值)
System.out.println( Math.sqrt( 8.6 ) ); //调用 Math 中的静态方法 sqrt(求平方根)
```

可以预先导入类中的静态成员，访问时省略类名，直接使用静态成员名，这样可以简化程序代码。例如：

```
import static Math.*;                   //在程序开头预先导入数学类 Math 里的所有静态成员
```

然后在程序中可以省略类名，直接使用静态成员名访问：

```
System.out.println( PI );               //访问静态字段 PI 时可省略类名
System.out.println( sqrt( 8.6 ) );      //调用静态方法 sqrt 时可省略类名
```

下面给出导入类中静态成员 import static 语句的语法。

Java 语法：import static **语句**

```
import static 包名.类名.静态成员名;
import static 包名.类名.*;
```

语法说明：

■ **import static** 语句用于导入某个类中的静态成员。其中，"**包名.类名.***"表示导入类中的所有静态成员。import static 语句可称为静态导入语句。

■ 通常，访问类中静态成员的形式应当是"包名.类名.**静态成员名**"。导入后可省略"包名.类名."，直接使用静态成员名访问，这样可以简化程序代码。

■ 导入后仍可以继续使用"包名.类名.静态成员名"的形式进行访问。

4. 包的绝对路径

包名对应的是某个根目录下的子目录名，即包名所表示的只是一个相对路径。在文件系统中查找包所对就应的子目录，这需要知道包的绝对路径(或称全路径)。Java 虚拟机会在哪些目录下查找包，或者说会将哪些目录作为包的根目录？

给定包名，Java 虚拟机会在以下两个目录中查找包：

(1) 运行 Java 虚拟机时选项"-classpath"所指定的目录。

(2) 环境变量 CLASSPATH 所指定的目录。

如果是在 Eclipse 集成开发环境中运行程序，Eclipse 会在 Java 项目属性所指定的组建路径(Java Build Path)中查找包。

3.5.4 访问权限

Java 语言中的访问权限控制分为两级：第一级先设定类的访问权限；第二级再设定类中成员的访问权限。可以看出，访问类成员会受到**类权限**和**成员权限**的两级控制。

1. 类的访问权限

类的访问权限有两种：公有权限和默认权限。

(1) **公有权限**：public。具有公有权限的类是**开放**的。使用 public 类不受控制。

(2) **默认权限**：定义时没有为类设定访问权限，则该类具有默认权限。具有默认权限的类是半开放的。具有默认权限的类只能被**本文件**或**本包**中的类使用。如果定义类时未指定访问权限，这意味着该类只能被**同一程序文件**或**同一目录**下其他程序文件中的类使用；任何其他目录(包括该类所在目录下的子目录)下的类都不能使用这个未指定访问权限的类。**默认权限**是向本文件或本包**定向开放**的一种权限。

2. 类成员的访问权限

类成员的访问权限有 4 种：公有权限、私有权限、默认权限和保护权限。

(1) **公有权限**：public。具有公有权限的类成员是**开放**的。访问 public 成员不受控制。

(2) **私有权限**：private。具有私有权限的类成员是**隐藏**的。private 成员只能在本类中访问，即只能被本类成员访问。在类外的任何其他地方都不能访问类的 private 成员。

(3) **默认权限**：定义时没有为类成员设定访问权限，则该类成员具有默认权限。默认权限是向本文件或本包定向开放的一种权限。具有默认权限的类成员除了能在本类中访问之外，还能被**本文件**或**本包**中的其他类访问。如果定义类成员时未指定访问权限，这意味着该类成员只能被**同一程序文件**或**同一目录**下其他程序文件中的类访问；任何其他目录(包括该类所在目录下的子目录)下的类都不能访问这个未指定访问权限的类成员。

(4) **保护权限**：protected。将在第 4 章讲解。

3.5.5 JAR 包

Java 程序开发结束后，可以将编译好的类程序文件(*. class)压缩打包成 Java **归档文件**(Java **AR**chive，简称为 **JAR**)。

使用 JDK 中的归档打包程序(jar. exe)对类程序文件及其附属文件(例如图片文件)进行压缩打包，所生成的归档文件扩展名为**. jar**。Java 归档文件俗称为 JAR 包。

1. 归档打包程序的使用

表 3-2 列出了与 JAR 包相关的 4 条常用命令，分别是创建、查看、解压或运行 JAR 包。

表 3-2 与 JAR 包相关的 4 条命令

功　　能	命　　令
创建 JAR 包	jar **cf** jar-file input-file(s)
查看 JAR 包中的内容	jar **tf** jar-file
解压 JAR 包，提取文件	jar **xf** jar-file
运行 JAR 包中的主类	**java** -jar app. jar

2. 打包举例

3.5.3 节图 3-22 曾给出一个项目 Project1 的完整目录结构和文件列表。下面就以这个项目为例来演示 JAR 包的使用。

使用归档打包程序 jar.exe，需先进入命令行界面(例如 Windows 操作系统的 cmd 窗口)，然后使用键盘命令 cd 将系统当前目录转到 Java 项目的类程序根目录。例如，使用如下命令将系统当前目录转到项目 Project1 的类程序根目录“d:\JavaTest\bin”：

```
cd  d:\JavaTest\bin <Enter 键>
```

1) 仅打包一个类程序文件

例如，将类程序根目录 bin 下的类程序文件 JClass1.class 打包成一个 JAR 包 myLib.jar，可使用如下命令：

```
jar cf  myLib.jar JClass1.class <Enter 键>
```

执行该命令，将在当前目录下生成一个 JAR 包文件 myLib.jar。可以使用如下命令来查看 JAR 包 myLib.jar 中的内容：

```
jar tf  myLib.jar <Enter 键>
```

上述两条命令的执行结果如图 3-23 所示。

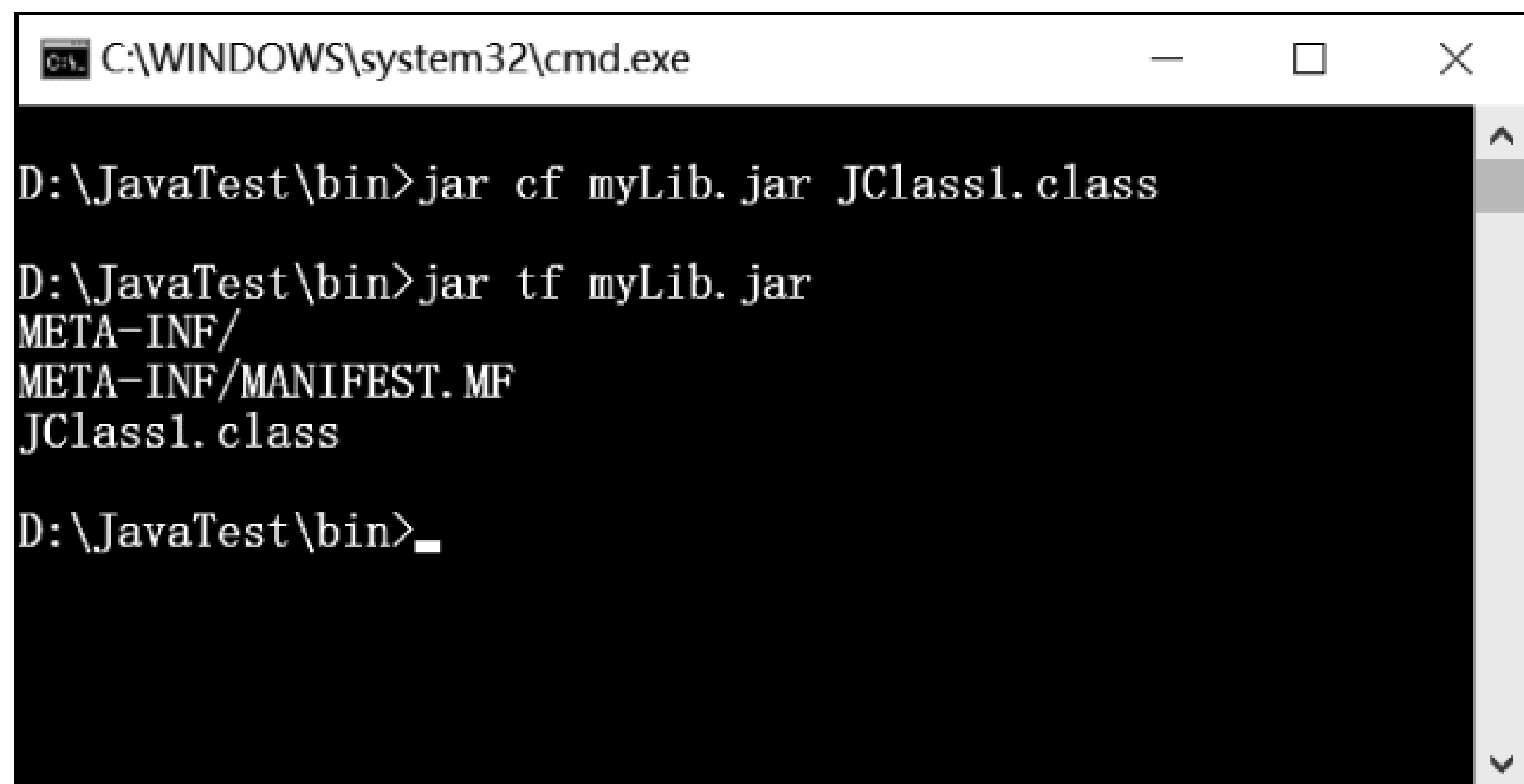


图 3-23　将 JClass1.class 打包成一个 JAR 包 myLib.jar

2) 打包一个子目录

例如，将子目录 lib1 下所有的类程序文件打包成一个 JAR 包 myLib1.jar，可使用如下命令：

```
jar cf  myLib1.jar lib1\*.class <Enter 键>
```

执行该命令，将在当前目录下生成一个 JAR 包文件 myLib1.jar，其中会包含一个子目录 lib1 和两个类程序文件 JClass2.class、JClass3.class。可以使用如下命令来查看 JAR 包 myLib1.jar 中的内容：

```
jar tf  myLib1.jar <Enter 键>
```

上述两条命令的执行结果如图 3-24 所示。

3) 打包一个可执行 JAR 包

例如，可以将项目 Project1，即类程序根目录 bin 下包括主类 JMainClass.class 在内的

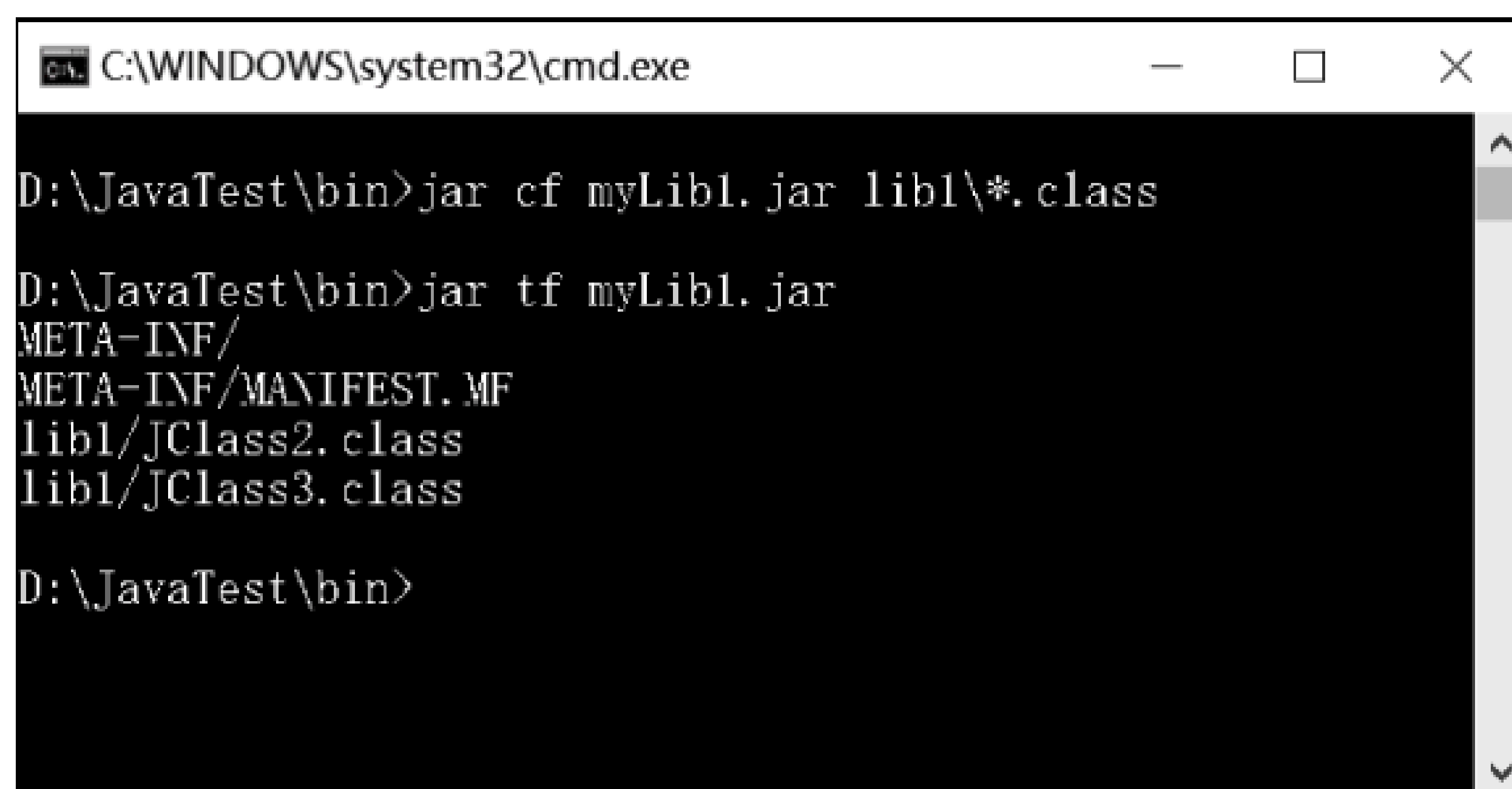


图 3-24 将子目录 lib1 下所有的类程序文件打包成一个 JAR 包 myLib1.jar

所有类程序文件,打包成一个 JAR 包 myApp.jar。因为主类 JMainClass.class 中定义了主方法 main(),因此所生成的 JAR 包 myApp.jar 是一个可以被 Java 虚拟机执行的 JAR 包。

打包可执行 JAR 包,需额外编写一个**清单**(manifest)文件,其目的是向 Java 虚拟机提供主类名、版本号等信息。使用文本编辑器(例如记事本)输入如下两行内容,并保存到一个文本文件 Manifest.txt 中。

```
Manifest-Version: 1.0
Main-Class: JMainClass
```

然后使用如下命令来生成可执行 JAR 包:

```
jar cfm  myApp.jar Manifest.txt  *.class lib1\*.class <回车键>
```

执行该命令,将在当前目录下生成一个 JAR 包文件 myApp.jar,其中会包含类程序根目录 bin(包括子目录 lib1)下的所有类程序文件。可以使用如下命令来查看 JAR 包 myApp.jar 中的内容:

```
jar tf  myApp.jar <回车键>
```

还可以使用如下命令启动 Java 虚拟机,执行 JAR 包 myApp.jar:

```
java - jar  myApp.jar <回车键>
```

上述 3 条命令的执行结果如图 3-25 所示。

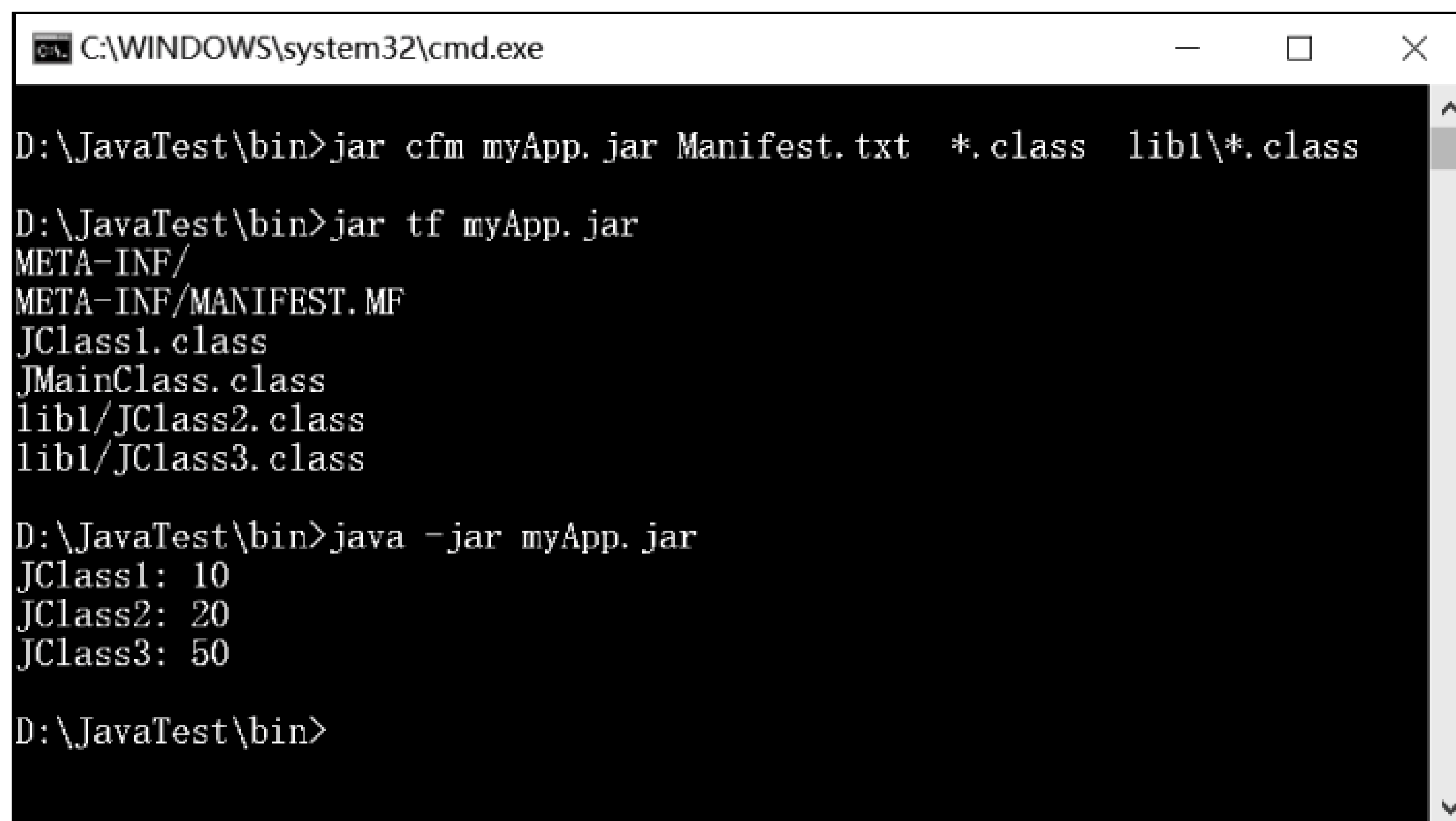


图 3-25 将项目 Project1 打包成一个可执行的 JAR 包 myApp.jar

本节习题

1. 下列关于 Java 程序文件的描述中，错误的是（　　）。
   A. 一个 Java 项目可以包含多个 Java 源程序文件
   B. 一个 Java 源程序文件中可以定义多个类，但其中最多只能有一个 public 类
   C. Java 源程序文件的扩展名是.java，类程序文件的扩展名是.class
   D. 编译后，一个 Java 源程序文件只会生成一个同名的类程序文件
2. 下列关于 Java 包的描述中，错误的是（　　）。
   A. 对 Java 类分包管理就是将类的程序文件放入不同子目录进行分组管理
   B. Java 类的包名就是其源程序文件所在的子目录名
   C. package 语句的作用是向 Java 编译器声明本文件中类所在的包名
   D. package 语句可以放在源程序代码的任意位置
3. 能够正确导入包 lib 中类 Circle 的语句是（　　）。
   A. import Lib.circle;　　B. import lib.?;
   C. import Circle;　　D. import lib.*;
4. 访问定义在 public 类中的 private 成员，下列访问形式中正确的是（　　）。
   A. 在本类中访问　　B. 在同一文件的类中访问
   C. 在同一包的类中访问　　D. 在不同包的类中访问
5. 访问定义在 public 类中的默认权限成员，下列访问形式中错误的是（　　）。
   A. 在本类中访问　　B. 在同一文件的类中访问
   C. 在同一包的类中访问　　D. 在不同包的类中访问
6. 访问定义在默认权限类中的 private 成员，下列访问形式中正确的是（　　）。
   A. 在本类中访问　　B. 在同一文件的类中访问
   C. 在同一包的类中访问　　D. 在不同包的类中访问
7. 访问定义在默认权限类中的默认权限成员，下列访问形式中错误的是（　　）。
   A. 在本类中访问　　B. 在同一文件的类中访问
   C. 在同一包的类中访问　　D. 在不同包的类中访问
8. 访问定义在默认权限类中的 public 成员，下列访问形式中错误的是（　　）。
   A. 在本类中访问　　B. 在同一文件的类中访问
   C. 在同一包的类中访问　　D. 在不同包的类中访问

本章学习要点

- 深入理解面向对象程序设计方法的基本原理和设计过程。
- 掌握 Java 语言中类与对象的语法规则。
- 理解引用数据类型与基本数据类型之间的区别。
- 掌握 Java 语言中与数组相关的语法。

• 掌握 Java 语言多文件结构的管理方法，重点理解包和子目录之间的对应关系。

本章习题

1. 编写程序。定义一个圆形类 Circle，其中包含 1 个私有字段成员(半径)，3 个公有方法成员(设置半径、计算面积和计算周长)和 3 个构造方法(不带参数的构造方法、带参数的构造方法和拷贝构造方法)。再定义一个主类测试圆形类 Circle，要求：

- 定义一个圆形对象 c1，然后从键盘输入一个数值 x 并将其设定为 c1 的半径，计算并显示 c1 的面积和周长。
- 再定义一个圆形对象 c2 并将半径初始化为 2x，计算并显示 c2 的面积和周长。
- 再定义一个圆形对象 c3 并用 c1 初始化 c3，计算并显示 c3 的面积和周长。

2. 编写程序。设计一个带日历的钟表类 Clock。编写类定义代码并测试这个类。

3. 编写程序。编写一个绘制正弦函数 sin(x)波形的 Java 程序。注：调用 Java 数学类 Math 的静态方法 sin(double a)求正弦值，其中 a 为以弧度为单位的角度。

4. 编写程序。编程 Java 程序，生成如下等差数列的前 10 项：$a_0 = 1, a_n - a_{n-1} = 3$，并保存到一个数组中。显示该数组的生成结果及其前 5 项之和。

5. 编写程序。模仿 3.4.3 节例 3-17 给出了一个具有可变长形参的求最小值方法的 Java 程序。

第4章 面向对象程序设计之二

面向对象程序设计提高程序开发效率的技术手段，主要有两个：一是分类管理程序代码；二是重用类代码。第3章已讲解了如何分类管理程序代码，即类与对象编程。本章将介绍如何重用类代码，重点讲解类的**组合**与**继承**。

本章还会深入讲解面向对象程序设计方法中的另外一个重要思想，即**多态**。多态性在字面上可理解为是一种程序代码的多义性。面向对象程序设计之所以提出多态的思想，其目的仍然是为进一步提高程序代码的重用性，进而提高软件开发和维护的效率。

4.1 重用类代码

程序＝数据＋算法。程序中的数据包括原始数据、中间结果和最终结果等。如何根据所处理的数据来合理使用和管理内存是编写程序的第一项工作内容。Java语言通过定义变量来申请内存。定义变量语句是与数据相关的代码，即**数据代码**。

将数据处理的过程细分成一组严格的操作步骤，这组操作步骤被称为算法。如何设计数据处理算法是编写程序的第二项工作内容。Java语言通过定义方法（即函数）来描述算法。方法是与算法相关的代码，即**算法代码**。

在面向对象程序设计中，类是重用“数据代码＋算法代码”的基本语法形式。重用类代码，可以同时重用其中的数据代码和算法代码，实现了对已有程序代码的完全重用，这极大地提高了程序开发效率。目前，面向对象程序设计方法是主流。

面向对象程序设计有3种重用类代码的形式，分别是用类定义对象、通过继承来定义新类或通过组合来定义新类。

4.1.1 用类定义对象

在面向对象程序设计中，程序员将相对独立、经常使用的功能提炼出来，编写成“类”这样可以重用的代码形式。

假设某位**程序员甲**将与钟表相关的程序功能提炼出来，用Java语言定义一个钟表类Clock。一个完整的类定义应包括4大要素，即字段成员、方法成员、构造方法以及各成员的访问权限。例4-1给出了一个钟表类Clock的完整定义代码。**注：**请读者记住这个类，本章后续讲解会经常用到这个类。

例 4-1 一个钟表类 Clock 的完整定义代码(Clock.java)

```
1  public class Clock {                          //定义钟表类 Clock
2      private  int hour, minute, second;       //字段:保存时、分、秒数据
3      public  void set(int h, int m, int s)    //方法:设置钟表对象的时间
4      {  hour = h;  minute = m;  second = s;  }
5      public  void show()                      //方法:显示时间,显示格式:时:分:秒
6      {  System.out.println( hour + ":" + minute + ":" + second );  }
7
8      public  Clock()                          //无参构造方法:将时分秒数据都设为 0
9      {  hour = 0;  minute = 0;  second = 0;  }
10     public Clock(int h, int m, int s)        //有参构造方法:根据参数设置时间
11     {  hour = h;  minute = m;  second = s;  }
12     public Clock( Clock oldObj )             //拷贝构造方法:复制已有对象的时、分、秒数据
13     {  hour = oldObj.hour;  minute = oldObj.minute;  second = oldObj.second;  }
14 }
```

假设,另一位**程序员乙**需要编写一个钟表相关的程序,他可以使用 Clock 类定义钟表对象。一个使用 Clock 类的典型流程如下:

```
Clock obj1 = new Clock();       //使用 Clock 类定义一个钟表对象 obj1
obj1.set( 8, 30, 15);           //调用 obj1 的公有方法 set(),设置其时间
obj1.show();                    //调用 obj1 的公有方法 show(),显示其时间,显示结果 8:30:15
```

用类定义对象,然后访问其成员,这实际上是在重用类的代码,实现其规定的程序功能。例如,程序员乙定义钟表类对象 obj1,然后调用其 set()、show()方法设置和显示时间,这就是在重用 Clock 类的代码,实现其规定的钟表功能。类代码是“**一次编写,长期使用**”。可以用钟表类 Clock 定义多个对象。例如:

```
Clock obj2 = new Clock( 9, 30, 15 );  //使用 Clock 类再定义一个钟表对象 obj2,定义时初始化
obj2.show();                       //调用 obj2 的公有方法 show(),显示其时间,显示结果 9:30:15
```

在上述重用钟表类 Clock 的过程中,有两种不同的程序员角色(见图 4-1)。**程序员甲**定义类,编写类代码;**程序员乙**使用类定义对象,然后通过对象重用类代码。用类定义对象,这是重用类代码的第一种形式。

4.1.2 用类继续定义新类

假设程序员乙不是简单地用 Clock 类定义钟表对象,而是要定义一个更加复杂的双时区钟表类 DualClock(见图 4-2)。经过分析,程序员乙抽象出双时区钟表的数据模型,其中包含两个钟表,用于表示两个不同的时区。这两个钟表都有自己的时、分、秒字段,也需要各自的设置、显示时间方法。

定义双时区钟表类 DualClock,程序员乙可以从零开始编写程序代码。但凭直觉我们就能知道,双时区钟表类 DualClock 和钟表类 Clock 有很多相似之处。程序员乙如果能够基于已有的钟表类 Clock 来设计双时区钟表类 DualClock,重用其中的某些代码,这样肯定能减少重复劳动,提高开发效率。

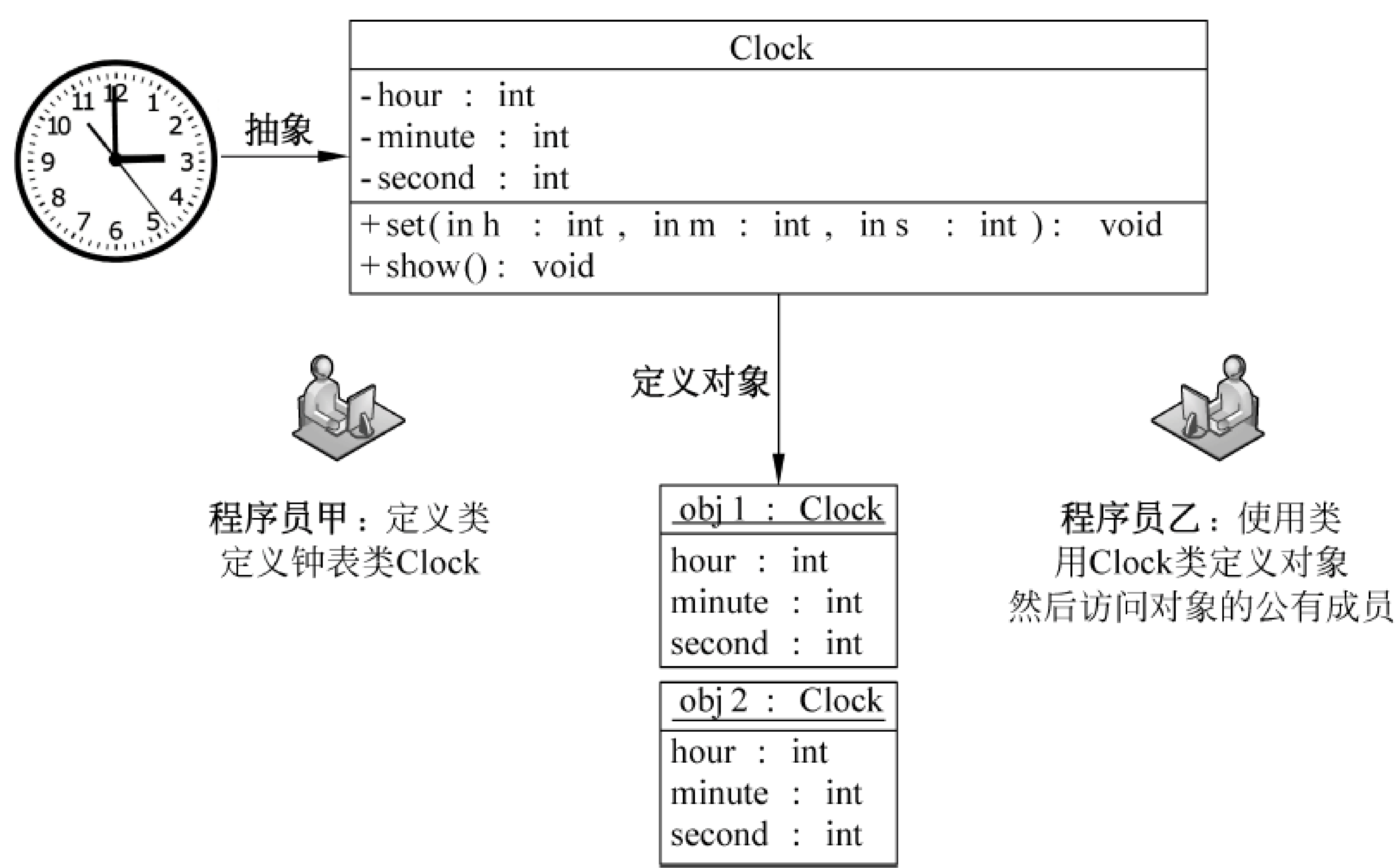


图 4-1　程序员乙重用 Clock 类代码：用类定义对象

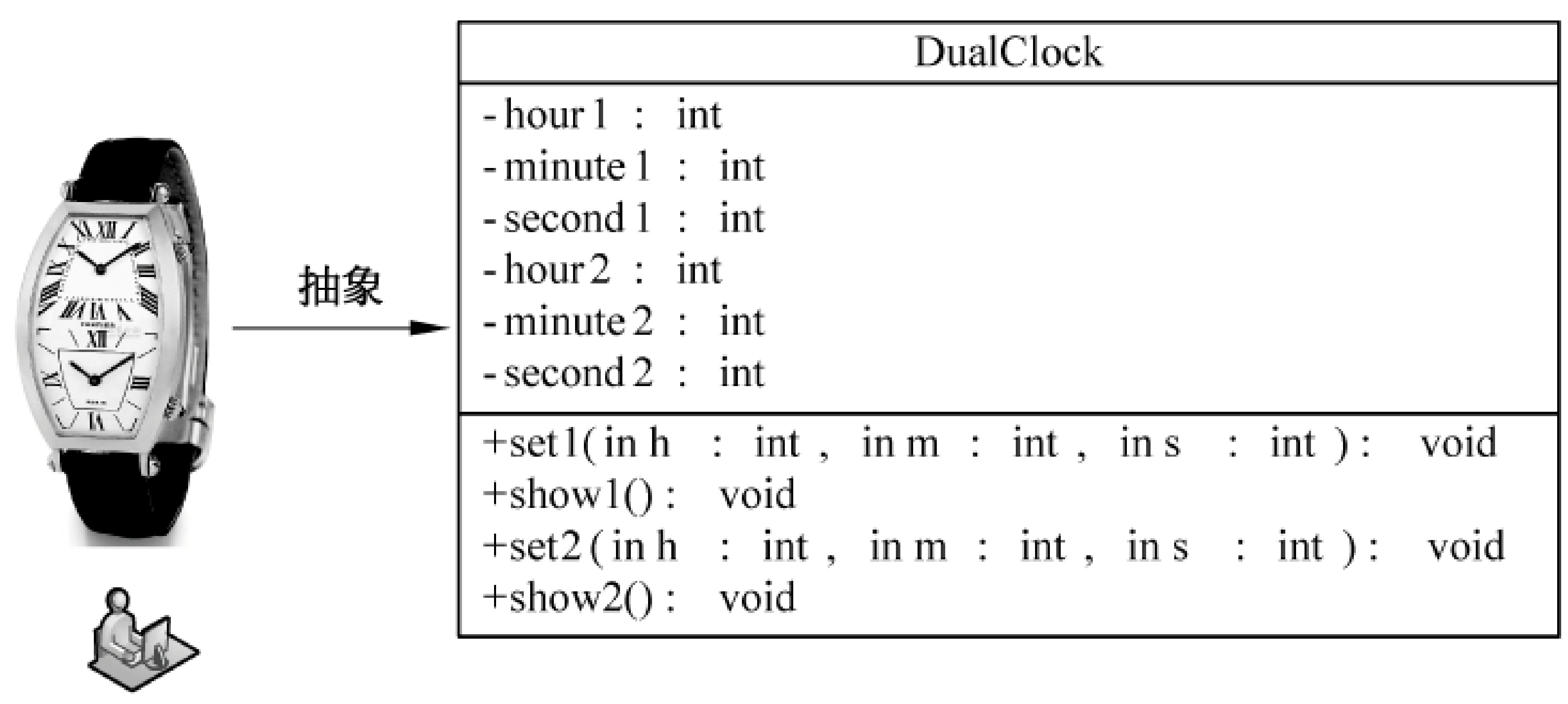


图 4-2　程序员乙需要定义一个描述双时区钟表的类 DualClock

如何充分利用已有的类来提高新类的开发效率，即如何重用已有类的代码来继续定义新类呢？针对这个问题，面向对象程序设计分别提出了组合和继承两种方法。Java 语言支持面向对象程序设计，为程序员使用组合或继承方法定义新类制定了完备的语法规则。

本节习题

1. 计算机程序由两个基本要素组成，这两个要素是(　　)。
   A. 程序和程序员　　B. 软件和硬件　　C. 数据和算法　　D. 类和对象
2. 结构化程序设计中调用函数，所重用的代码是(　　)。
   A. 程序员　　　　B. 数据代码
   C. 算法代码　　　D. 数据代码＋算法代码

3. 结构化程序设计中使用结构体定义变量,所重用的代码是(　　)。
   A. 程序员　　B. 数据代码
   C. 算法代码　　D. 数据代码+算法代码
4. 面向对象程序设计中使用类定义对象,所重用的代码是(　)。
   A. 程序员　　B. 数据代码
   C. 算法代码　　D. 数据代码+算法代码
5. 面向对象程序设计重用类代码有不同的形式,其中不包括(　　)。
   A. 用类定义对象　　B. 类的组合　　C. 类的继承　　D. 复制类代码

4.2 类的组合

类不是Java语言预定义的基本数据类型,而是由多个基本类型字段组合在一起形成的**自定义**数据类型。用简单的零件组装复杂的整体是人们常用的一种方法,例如计算机工厂将主板、CPU、内存条、硬盘等零件组装在一起生产出整机。零件通常不是自己生产,而是从专门厂家购买来的,这样可以降低生产难度,提高效率。

程序员可以将别人编写的类当作零件(**零件类**),在此基础上定义自己的新类(**整体类**),这就是类的组合。组合的**编程原理**是:程序员在定义新类的时候,使用已有的类来定义字段。这些字段是类类型的对象,称为**对象字段**。Java语言将包含对象字段的类称为**组合类**。按照数据类型的不同,组合类中字段成员可分为两种,即类类型的对象字段和基本数据类型的非对象字段。

使用组合类定义对象,即**组合类对象**,其字段成员中将包含对象字段和非对象字段。访问组合类对象中的非对象字段,其访问形式与第3章所介绍的访问字段成员没有区别,即:

组合类对象名.非对象字段名

而组合类对象中的对象字段还包含自己的下级成员,也就是说组合类对象包含多级成员。访问组合类对象中的对象字段,可以继续访问其下级成员,这是一种多级访问。多级访问的语法形式是:

组合类对象名.对象字段名.对象字段的下级成员名

4.2.1 组合类的定义

假设给定程序员甲定义的钟表类Clock,在此基础上程序员乙定义一个双时区钟表类DualClock。双时区钟表类DualClock可认为是由两个Clock类的对象组合而成的,图4-3演示了程序员乙使用钟表类Clock定义组合类DualClock的过程。

图4-3中,使用钟表类Clock定义组合类DualClock,就是重用钟表类Clock的代码。在这个重用过程中有两个程序员角色:一是提供钟表类Clock的程序员甲;另一个是使用该类继续定义新类DualClock的程序员乙。例4-2给出了基于钟表类Clock,使用组合方法定义双时区钟表类DualClock的Java示意代码。

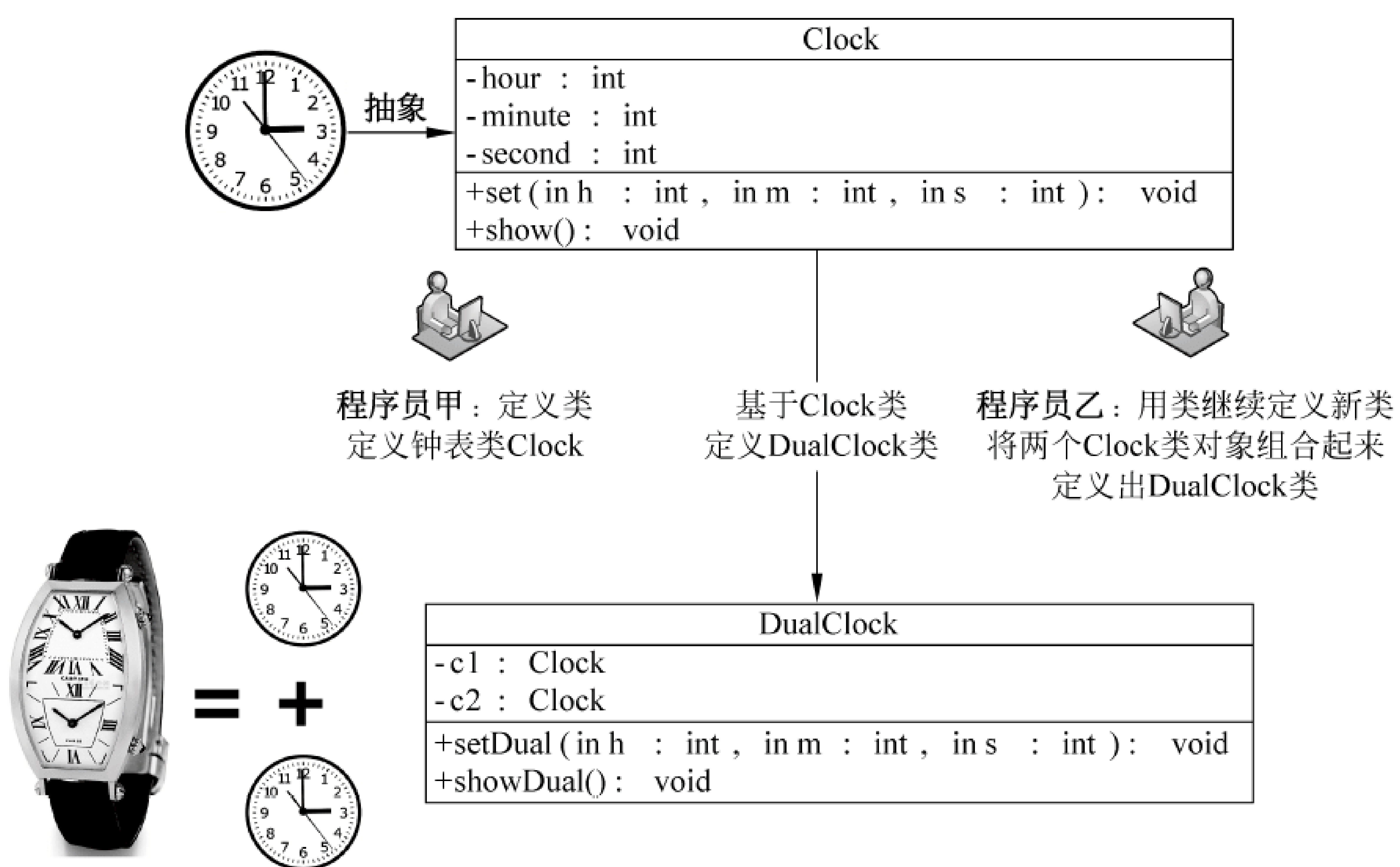


图 4-3　程序员乙重用 Clock 类代码：用类定义组合类

例 4-2　一个使用钟表类 Clock 组合出的双时区钟表类 DualClock(DualClock.java)

```
1  publicclass DualClock {                //双时区钟表类:含有对象字段,属于组合类
2      public Clock c1, c2;               //对象字段:两个 Clock 类的钟表对象,设为公有成员
3      public void setDual(int h, int m, int s)    //设置方法:按参数设置 c1、c2 的时间
4      {  c1.set( h, m, s );  c2.set( h +1, m, s);  } //假设设为两个连续的时区
5      public void showDual()                      //显示两个钟表的时间
6      {  c1.show();  c2.show();  }
7
8      public DualClock() {               //组合类需要定义自己的构造方法
9          c1 = new Clock();              //组合类构造方法需使用运算符 new 创建对象字段
10         c2 = new Clock();
11     }
12 }
```

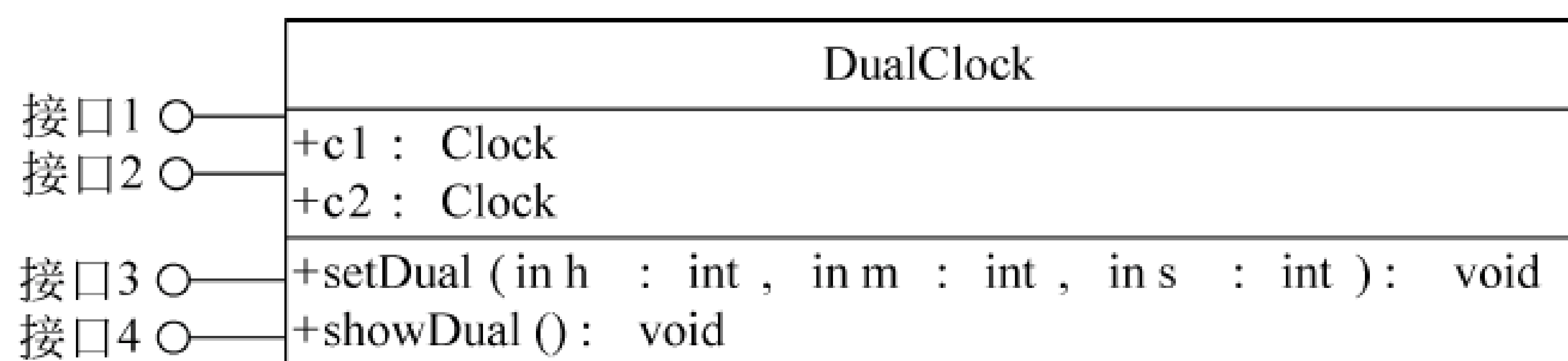
组合类中对象字段的语法细则如下。

(1) 在组合类的方法成员中访问对象字段。

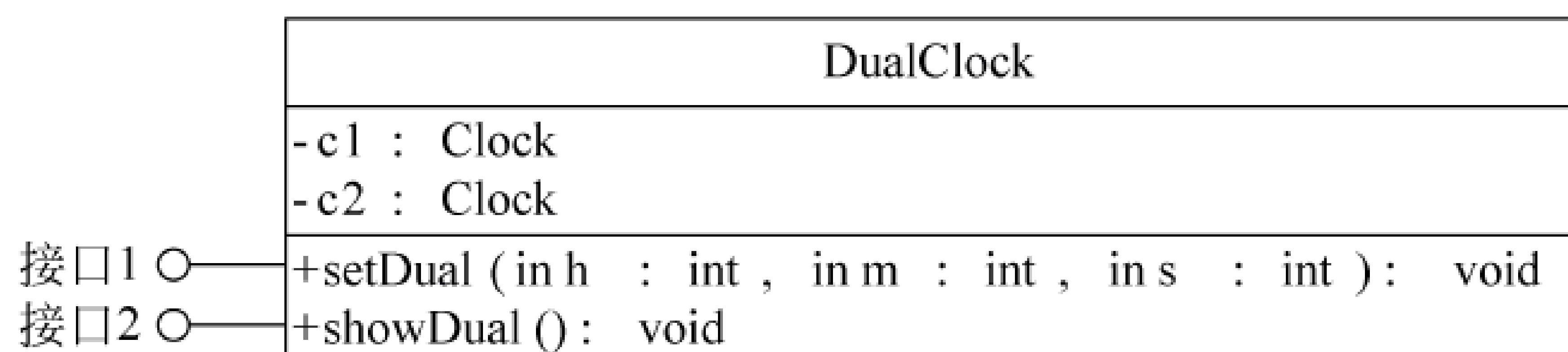
组合类中的方法在处理对象字段时，会访问其下级成员。例如，例 4-2 组合类 DualClock 中的设置时间方法 setDual()，会分别调用对象字段 c1、c2 的下级成员 set()方法。组合类中的方法在访问对象字段的下级成员时，会受到这些下级成员的访问权限控制。

(2) 对象字段的二次封装。

例 4-2 中的组合类 DualClock 将对象字段 c1、c2 的访问权限设定成 **public**，就是在组合类中开放这两个对象字段，如图 4-4(a)所示。如果将对象字段 c1、c2 的访问权限设为 **private**，则对象字段及其下级成员都将被隐藏起来，如图 4-4(b)所示。为组合类中的对象字段设定访问权限，实际上是对它们进行二次封装。



(a) 开放对象字段c1和c2



(b) 隐藏对象字段c1和c2

图 4-4 组合类中对象字段的二次封装

4.2.2 组合类对象的定义与访问

1. 定义组合类对象

与任何普通的类一样,可以使用组合类来定义对象。例如,定义一个组合类 DualClock 的对象 obj:

```
DualClock obj = new DualClock();          //定义一个组合类 DualClock 的对象 obj
```

计算机执行该对象定义语句,将在内存中创建一个组合类 DualClock 的对象,为其中的字段成员分配内存空间(如图 4-5 所示)。可以看出,一个组合类 DualClock 的对象实际上只包含两个引用变量 c1 和 c2,然后再由这两个引用变量去引用具体的钟表对象。

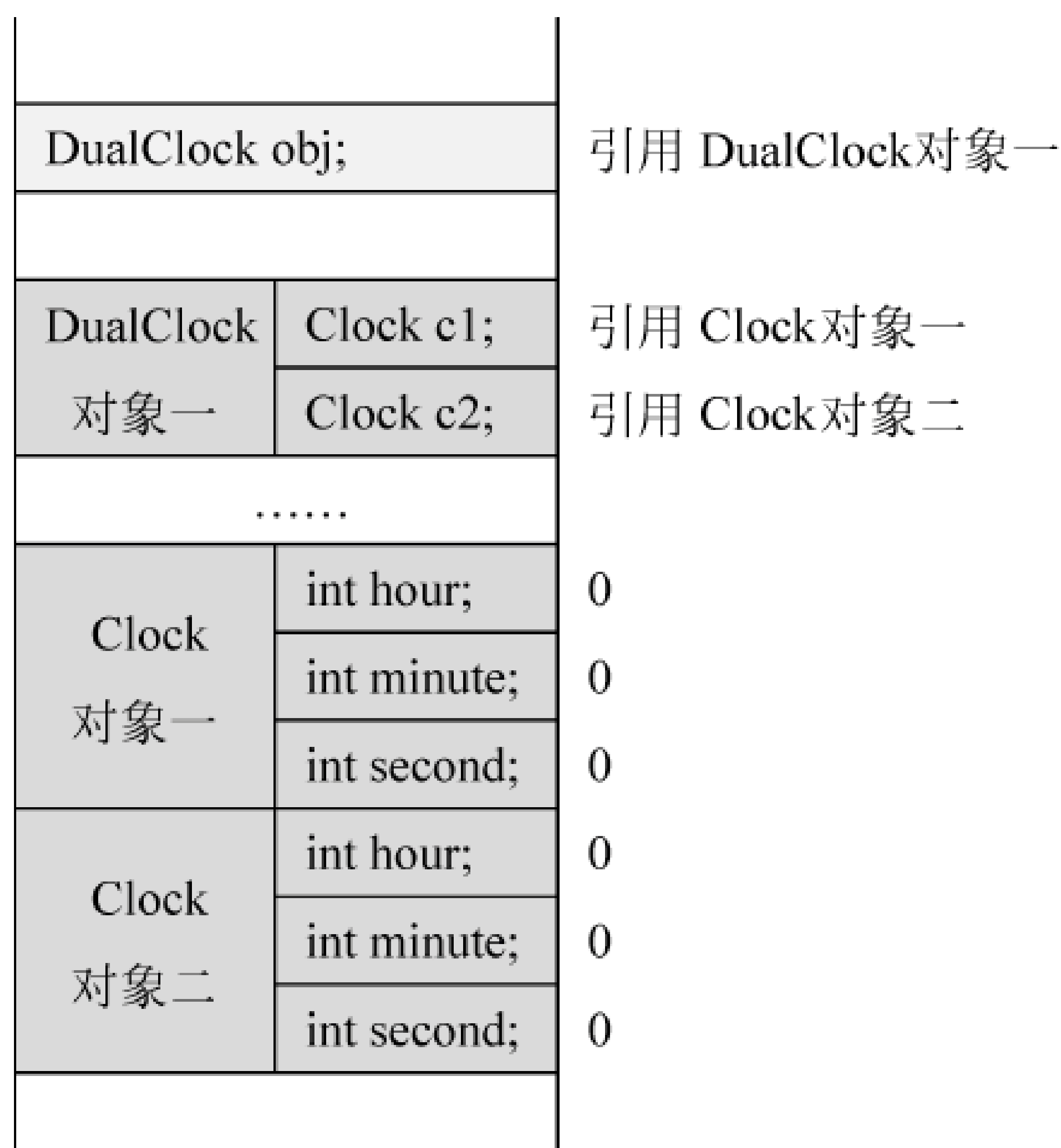


图 4-5 组合类对象 obj 的内存分配

2. 访问组合类对象及其下级成员

按照组合类 DualClock 的定义,对象 obj 将包含两个对象字段 c1、c2,还有一个设置时

间方法 setDual()和一个显示时间方法 showDual()，另外还有一个构造方法，总共 5 个成员。除了构造方法是在创建对象时自动调用外，程序员可以访问对象 obj 的其他 4 个成员。这 4 个成员的访问形式分别是：obj. **c1**、obj. **c2**、obj. **setDual**()、obj. **showDual**()。

与普通对象不同的是，组合类对象含有对象字段。对象字段还包含下级成员，可以访问这些下级成员，这就是组合类对象的多级访问。例如：

```
obj.c1.set( 10, 15, 30 );          //设置对象 obj 第一个钟表 c1 的时间
obj.c1.show();                     //显示对象 obj 第一个钟表 c1 的时间，显示结果:10:15:30
```

访问对象字段的下级成员时，会受到其访问权限控制。例如，下列设置对象字段 c1 时间的方法是错误的：

```
obj.c1.hour   = 10;          //设置对象 obj 第一个钟表 c1 的小时数:错误,hour 是私有权限
obj.c1.minute = 15;          //设置对象 obj 第一个钟表 c1 的分钟数:错误,minute 是私有权限
obj.c1.second = 30;          //设置对象 obj 第一个钟表 c1 的秒数:错误,second 是私有权限
```

访问组合类对象中对象字段的下级成员，是一种**多级访问**。多级访问会受到**多级权限**的控制。访问组合类对象中对象字段的下级成员，首先要求对象字段是可访问的，同时还要求其下级成员也是可访问的，这两个条件必须同时满足。

3. 如何设计组合类中对象字段的访问权限

组合类将其他类(零件类)的对象作为自己的字段成员，相当于是用零件来组装产品。用零件组装产品时要考虑，是将零件直接暴露给用户，还是将零件隐藏起来。这个问题要根据产品及零件的功能来决定。例如，计算机厂家在组装计算机时会根据功能要求，将用户不需要直接操作的主板、CPU、内存条和硬盘等零件隐藏起来，即用一个机箱将这些零件"封装"起来；而将用户使用计算机所必需的电源开关、键盘、鼠标、光盘和显示器等零件开放出来，放在机箱外面。

在组合类 DualClock 的定义和使用过程中总共有 3 个程序员角色：**程序员甲**是编写钟表类 Clock 的程序员；**程序员乙**是编写组合类 DualClock 的程序员；**程序员丙**是使用 DualClock 类定义组合类对象 obj 的程序员(见图 4-6)。

程序员乙通过定义 Clock 类的对象字段来组装 DualClock 类。组装时，程序员乙应根据功能要求决定将哪些对象字段开放给程序员丙，哪些应当隐藏起来。开放就是将对象字段设定为公有权限，隐藏就是将其设定为私有权限，这就是程序员乙在定义组合类时为对象字段所做的二次封装。

对象字段还包含下级成员，这些下级成员也都有各自的访问权限，它们是程序员甲在定义 Clock 类时就已经设定好了的。在对象字段的下级成员中，哪些成员程序员丙可以访问，哪些不能访问，这不是程序员乙设定的，而是程序员甲在定义 Clock 类时就已经确定了的。

4. 多级组合

使用零件组装出的产品可以继续作为零件去组装更大的产品，组装可以任意多级。例如，计算机厂家用零件组装计算机，而 ATM 机厂家又会把计算机作为零件来组装 ATM 机。组装 ATM 机时会进行再次封装，例如将计算机的显示器继续开放出来，但把 ATM 机

用户不需要的键盘、鼠标、光盘等接口都封装起来。

组合类也可以任意多级。用零件类定义组合类，组合类可以继续作为零件类去定义更大的组合类，这就是**多级组合**。多级组合过程中，每一级组合类都会根据自己的功能需要设定对象字段的访问权限，这就产生了**多级封装**。

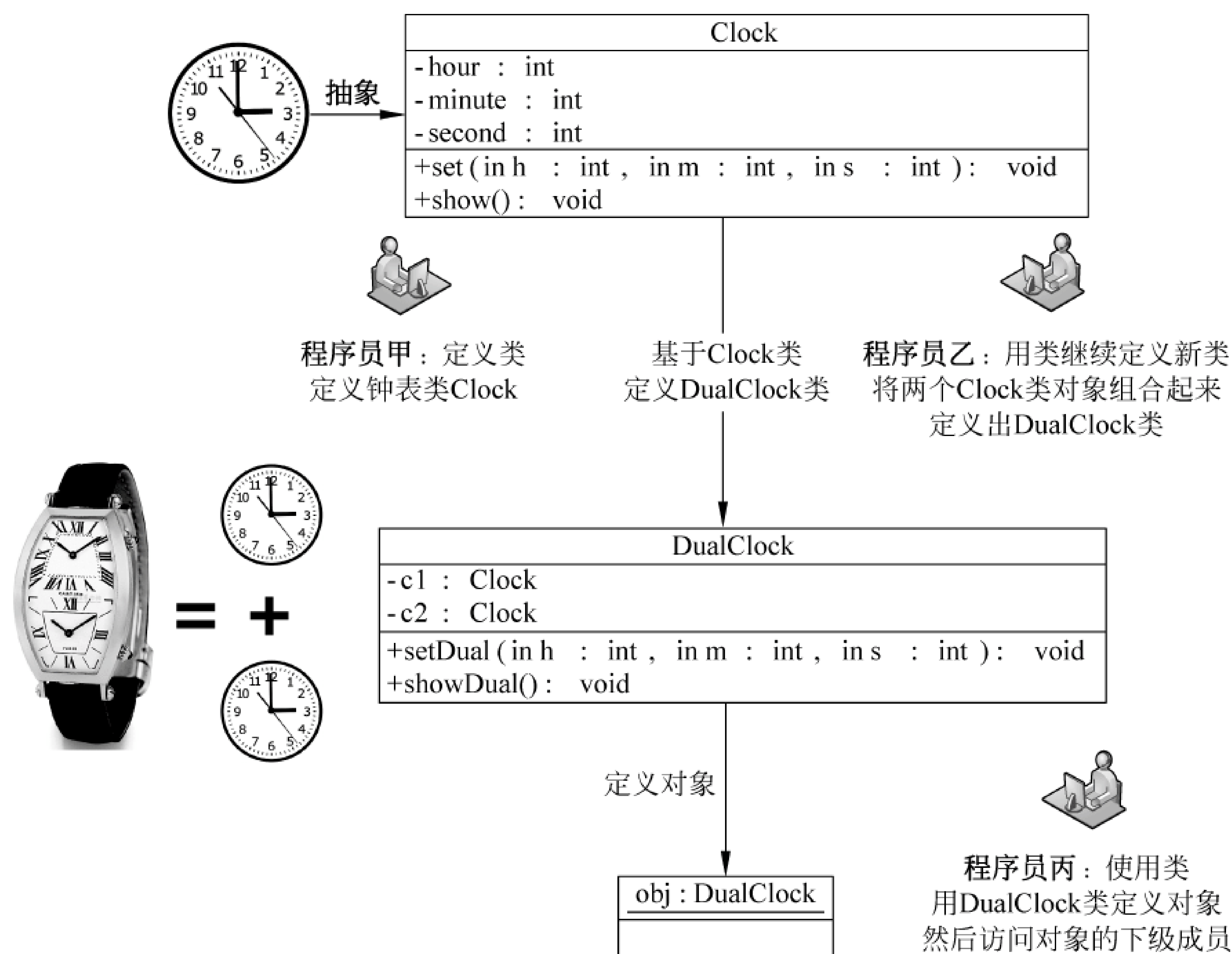


图 4-6 组合类 DualClock 在定义和使用过程中的 3 个程序员角色

4.2.3 组合类的构造方法

计算机执行定义对象语句时将创建对象，为其分配内存空间，并自动调用对象所属类的构造方法来初始化对象，这个过程就是对象的构造。

按照数据类型的不同，组合类中字段成员可分为两种，即类类型的对象字段和基本数据类型的非对象字段。创建组合类对象，实际上只为其中的对象字段定义了一个引用变量。该引用变量被初始化为空引用，还没有引用某个具体的对象。组合类需要在设计构造方法时考虑如何为对象字段创建对象。为对象字段创建对象有 4 种方法。

1. 在构造方法中为对象字段创建对象

可以在构造方法中为对象字段创建对象。例如，例 4-2 所示的组合类 DualClock 就是在构造方法中(代码第 9～10 行)为对象字段创建对象的。

```
c1 = new Clock();          //使用运算符 new 为对象字段创建对象
c2 = new Clock();
```

2. 在定义对象字段时直接创建对象

可以在类中定义对象字段时直接创建对象。例如，可以将例 4-2 中第 2 行的定义对象字段语句修改为如下形式：

```
public  Clock c1 = new Clock(), c2 = new Clock();
```

该语句就是在定义对象字段 c1、c2 时直接创建对象。**注**：这时应删除构造方法里的创建对象语句(例 4-2 中的第 9～10 行)。

3. 向构造方法传递已经创建好的对象

可以为组合类 DualClock 添加一个如下的构造方法：

```
public DualClock( Clock p1, Clock p2) {    //向构造方法传递两个已经创建好的钟表对象
    c1 = p1;                               //对象字段 c1 将直接引用所传递过来的钟表对象 p1
    c2 = p2;                               //对象字段 c2 将直接引用所传递过来的钟表对象 p2
}
```

这时，使用组合类 DualClock 定义对象，可以先创建好两个钟表对象，然后再传递给构造方法。

```
Clock cObj1 = new Clock(), cObj2 = new Clock();    //先创建好两个钟表对象 cObj1 和 cObj2
DualClock obj = new DualClock( cObj1, cObj2 );     //定义组合类对象时再传递给构造方法
```

这两条语句可简写为如下形式：

```
DualClock obj = new DualClock( new Clock(), new Clock() );
```

4. 直接引用其他组合类对象的对象字段

可以为组合类 DualClock 添加一个如下的拷贝构造方法：

```
public DualClock( DualClock p) {    //向构造方法传递一个已有的组合类对象
    c1 = p.c1;                      //对象字段 c1 将直接引用所传递过来组合类对象 p 的 c1
    c2 = p.c2;                      //对象字段 c2 将直接引用所传递过来组合类对象 p 的 c2
}
```

这时，使用组合类 DualClock 再定义一个新对象 obj1，可以按如下形式向构造方法传递前面已经定义好的组合类对象 obj：

```
DualClock obj1 = new DualClock( obj );  //定义组合类对象 obj1 时传递前面已经定义好的 obj
```

这样所定义出的组合类对象 obj1 将和 obj 一起，共用两个相同的钟表对象。

4.2.4　包装类

可以对一个已有的类进行重新**包装**(wrap)，其目的是调整或增强类的功能。包装的方法就是将一个已有类的对象作为字段，然后调整或增强其功能。包装后的类称作已有类的**包装类**。从本质上讲，包装类就是一个组合类。

例如,可以对钟表类 Clock 重新包装,将其增强为一个带日历的包装类 DateClock。例 4-3 给出了完整的示例和测试代码。

例 4-3 对钟表类 Clock 重新包装得到一个带日历功能的包装类 DateClock

```
 1  public class DateClock {                   //包装类(DateClock.java)
 2      private Clock c;                       //对象字段:被包装的原始钟表(Clock)对象 c
 3      //以下代码都是为了对钟表对象 c 进行包装,为其增加日历功能
 4      private int year, month, day;          //添加字段:保存年、月、日数据
 5      public void setDate(int y, int m, int d)//方法:设置日期
 6      {year = y;  month = m;  day = d;  }
 7      public void show() {                   //方法:显示日期和时间
 8          System.out.print(year +"-" +month +"-" +day +" ");    //先显示日期
 9          c.show();                          //再显示时间
10      }
11      public Clock getClock()                //方法:获得包装前的原始钟表对象 c
12      {return c;}
13      public DateClock( Clock obj)           //构造方法:传递被包装的钟表对象
14      {  c = obj;  }                         //对象字段 c 直接引用传递过来的钟表对象 obj
15  }
 1  public class DateClockTest {               //主类(DateClockTest.java)
 2
 3      public static void main(String[] args) {  //主方法
 4          Clock cObj = new Clock(10, 30, 15 );    //定义一个钟表对象 cObj
 5          //对钟表对象 cObj 进行包装,得到一个带日历的钟表对象 dcObj
 6          DateClock dcObj = new DateClock( cObj );
 7          dcObj.setDate(2018, 9, 1);         //设置 dcObj 的日期
 8          dcObj.show();           //显示 dcObj 的日期和时间,显示结果 2018-9-1 10:30:15
 9      }
10  }
```

例 4-3 中,包装类 DateClock 的作用就是对钟表对象 cObj 进行包装,得到一个功能更强的新对象 dcObj。新对象 dcObj 是一个带日历功能的钟表。

组合类小结如下。

(1) **代码重用**。组合是一种有效的重用代码形式。程序员在设计新类时应首先了解一下有哪些可以重用的类。这些类可以是自己以前编写的,或是 JDK 提供的,或是从市场上购买来的。可根据功能需要,采用组合的方法来设计新类。

(2) **多级组合**。用零件类定义组合类,组合类可继续作为零件类去定义更大的组合类,这就是类的多级组合。多级组合是一种"自底向上"的程序设计方法。类越往上组合,其功能就越多。

(3) **多层封装**。多级组合过程中,每一级组合类都会根据自己的功能需要设定对象字段的访问权限。有多少级组合,就会有多少层封装。

(4) **包装类**。定义包装类的目的是增强或调整已有类的功能。包装也可以任意多级,即多级包装。包装类是组合类的一个特例。

本节习题

1. 下列关于组合类的描述中，正确的是(　　)。

   A. 字段成员中包含类类型的对象字段，这样的类被称为组合类

   B. 方法成员访问了类类型对象的字段成员，这样的类被称为组合类

   C. 方法成员调用了类类型对象的方法成员，这样的类被称为组合类

   D. 组合类字段成员中不能包含非对象字段，即用基本数据类型定义的字段

2. 下列关于组合类对象字段的描述中，错误的是(　　)。

   A. 所谓对象字段，就是用类定义的对象

   B. 对象字段还包含下级成员

   C. 组合类设定对象字段的访问权限是对其进行二次封装

   D. 组合类中的方法成员访问对象字段的下级成员不受权限控制

3. 下列关于组合类对象的描述中，错误的是(　　)。

   A. 组合类所定义的对象中包含对象字段

   B. 访问组合类对象中对象字段的下级成员是多级访问

   C. 访问组合类对象中对象字段的下级成员需受多级权限的控制

   D. 可以访问组合类对象中 private 对象字段的下级 public 成员

4. 定义如下的类 A 和组合类 B：

```
class A {
    private   int x;
    public   int y;
}
class B {
    public   A t;
    public   int s;
}
```

使用组合类 B 定义一个对象 obj，则下列语句中正确的是(　　)。

   A. obj.x = 5;　　obj.y = 5;　　obj.s = 5;

   B. obj.t.x = 5;　　obj.t.y = 5;　　obj.t.s = 5;

   C. B x = obj;　　x.y = 5;　　x.s = 5;

   D. B y = obj;　　y.t.y = 5;　　y.s = 5;

5. 下列关于组合类构造对象字段的描述中，错误的是(　　)。

   A. 组合类可以在构造方法中为对象字段创建对象

   B. 组合类可以在类中定义对象字段时直接创建对象

   C. 定义组合类对象时可以向构造方法传递已经创建好的对象

   D. 不同组合类对象的对象字段不能共用对象，即不能引用同一个对象

4.3 类的继承与扩展

遗传与变异是生物进化的基础，是继承祖先优良品质的同时不断进化以适应新的生存环境的重要机制。面向对象程序设计借鉴了这种机制，为类代码提供了一种新的，也是最为重要的一种代码重用形式，这就是类的继承与扩展。

程序员面对新的程序设计问题可能需要设计新的类。设计新类时可以**继承**(inherit)已有的类，这个已有的类被称为**超类**(super class)或父类。继承的目的是重用已有类的代码，从而提高新类的开发效率。

如果仅仅是单纯继承超类，那就只是简单的克隆，在程序设计中没有实际意义。在继承超类的基础上进行**扩展**(extend)，或者对从超类继承来的功能进行**重新定义**(override，又称为**覆盖**或**重写**)，这样所得到的新类被称为**子类**(subclass)。子类可以解决新的程序设计问题。

继承与扩展的**编程原理**是：程序员在定义新类时，首先继承已有超类的字段和方法；在此基础上进行扩展，例如添加新成员，或重写从超类继承来的成员，这样所定义出的新类就是子类。子类具有超类的全部功能，同时还扩展或完善了某些新功能。

按照来源的不同，子类中的成员可分为两种：一种是从超类继承来的成员，称为**超类成员**；另一种是定义时新添加或重写的新成员，称为**子类成员**。

4.3.1 子类的定义

Java 语法：**定义子类**

```
[public ] class 子类名 extends 超类名 {
    …        //新添加的成员
    …        //重新定义的成员
}
```

语法说明：

- 定义**子类**(新类)时，需使用关键字 **extends** 指定所继承的**超类**(已有类)，然后在此基础上进行扩展。一个类只能继承一个超类，即类只能单继承。
- 子类将**继承**超类中的所有字段和方法(静态成员、构造方法除外)，子类不用编写任何代码就能拥有与超类相同的字段和方法。子类中从超类继承来的成员被称为**超类成员**。超类成员会保持其原有的访问权限和功能。
- 超类中的**静态成员**虽然未被子类继承，但可通过子类名或子类对象访问它们。静态成员可认为是被本类及其所有子类对象共享的成员。
- 子类可以**添加**新字段或新方法，这样就能扩展超类中没有的功能。子类新添加的成员被称为是**子类成员**。
- 子类可以**重新定义**(重写)超类成员，即添加与超类成员同名的字段，或具有相同签名的方法。访问这些成员，子类成员将**覆盖**(override))重名的超类成员。重名的超

类成员依然存在，但它们被屏蔽了。实际应用主要是重写超类继承来的方法，重写的目的是**替换**或**增强**原有方法的功能。虽然语法上可以重新定义超类继承来的字段，但没有什么实际用途，而且会造成数据混乱，建议尽量不使用。

■ 可以访问被覆盖的超类成员，这时需要使用关键字 **super** 来明确指定超类成员，例如 super.字段名、super.方法名()。关键字 super 代表超类。**注**：只能在子类新添加的方法成员中使用 super 关键字访问超类成员。

假设要定义一个手表类 Watch，经过分析可以抽象出手表的数据模型（见图 4-7），其中包括保存时、分、秒数据（hour、minute、second）和表带类型（band，金属表带或皮革表带）的字段，共 4 个字段；还包括设置时间方法 set()、显示时间方法 show() 和设置表带类型的方法 setBand()，共 3 个方法。

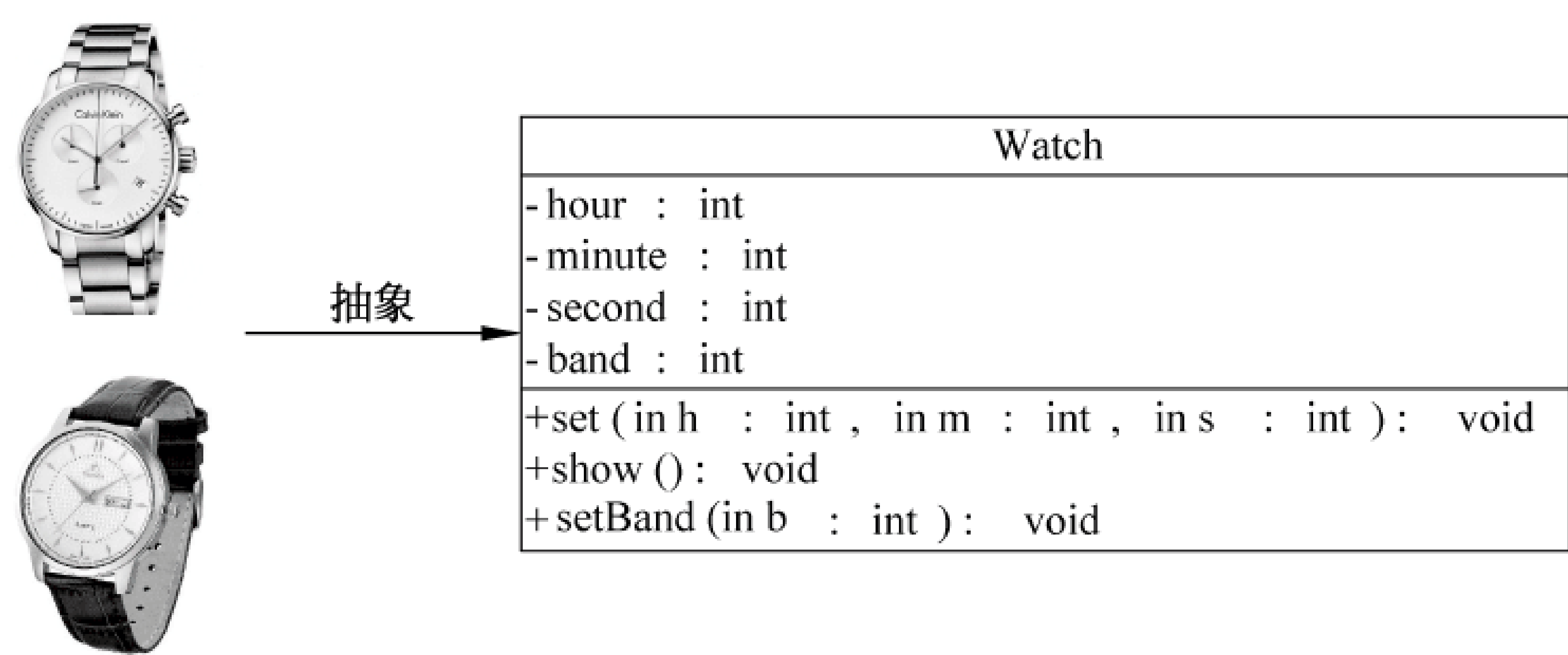


图 4-7　手表类的数据模型

从图 4-7 可以看出，手表类 Watch 实际上就是一种钟表，与钟表类 Clock 有很多重复的内容。定义手表类 Watch 可以直接继承钟表类 Clock 中的某些成员，例如时、分、秒字段 hour、minute、second，以及设置时间的方法 set()，然后在此基础上进行扩展，例如添加表带类型字段 band、设置表带类型的方法 setBand()等。

给定钟表类 Clock，通过继承与扩展来定义手表类 Watch，这样可以减少很多重复代码，有效提高开发效率。在这个继承与扩展过程中，钟表类 Clock 是已有的超类，手表类 Watch 则是新定义的子类。例 4-4 给出手表类 Watch 的完整定义代码。

例 4-4　通过继承与扩展钟表类 Clock 所定义出的手表类 Watch(Watch.java)

```
1  publicclass Watch extends Clock {  //继承超类 Clock,在此基础上扩展出子类 Watch
2      private int band = 1;          //新添加的字段:表带类型,1 - 金属,2 - 皮革
3      public void setBand(int b)     //新添加的方法:设置表带类型
4      {  band = b;  }
5      public void show() {         //重新定义显示时间的方法,显示格式:(表带类型)时:分:秒
6          //先显示表带类型
7          if (band == 1) System.out.print("(金属表带)");
8          else System.out.print("(皮革表带)");
9          //再显示时间:调用超类的方法 show()
10         super.show();              //关键字 super 表示超类
11     }
12 }
```

定义子类的语法细则如下。

(1) 子类继承超类的成员。

子类会继承超类的所有字段和方法(静态成员、构造方法除外)。在例 4-4 中,子类 Watch 会继承下列超类 Clock 的 5 个成员:

```
private int hour, minute, second;          //字段:保存时分秒数据
public void set(int h, int m, int s);      //方法:设置钟表对象的时间
public void show();                        //方法:显示时间,显示格式:时:分:秒
```

在子类 Watch 中,这 5 个继承来的成员被称为**超类成员**。继承到子类 Watch 后,这 5 个成员具有与在超类 Clock 时完全相同的访问权限和功能。继承超类成员就是重用其代码,类似于是将它们的代码从超类复制到子类中来。

(2) 子类添加新成员或重写超类成员。

子类会在继承超类的基础上进行扩展,例如添加新成员,或重写从超类继承来的成员,这些成员统称为**子类成员**。在例 4-4 中,子类 Watch 为了描述手表的表带属性,新添加了一个表示表带类型的字段 band,以及一个设置表带类型的方法 setBand()。

```
private  int band;                         //新添加的字段:表带类型,1 - 金属,2 - 皮革
public void setBand(int b);                //新添加的方法:设置表带类型
```

由于增加了表带属性,子类 Watch 希望在显示时间时能同时显示出表带类型。为此,子类 Watch 又添加了一个显示时间的方法 show()。

```
public void show();            //重新定义显示时间的方法,显示格式:(表带类型)时:分:秒
```

注意:这个新添加的方法 show()与从超类 Clock 继承来的超类方法 show()具有相同的签名(即调用接口)。添加与超类方法具有相同签名的方法,这实际上是在子类中**重新定义**该方法的功能,或称为对超类方法的**重写**。

重写超类方法的目的是**替换**或**增强**原有方法的功能。例如子类 Watch 重写的方法 show()在原有显示时间的基础上又增加了显示表带类型的功能。

(3) 子类成员访问超类成员。

在子类定义新的方法成员时可能需要访问继承来的超类成员,这种访问形式就是子类成员访问超类成员。子类成员可以访问超类成员,但访问时会受到其访问权限的控制。例如在例 4-4 中,子类 Watch 继承来的超类字段 hour、minute 和 second 是私有的,在定义新的显示时间方法 show()时不能直接访问它们,而必须通过继承来的公有超类方法 show()才能显示出时、分、秒数据。

如果子类重写了超类成员,则子类将包含两个重名的成员:一个是继承来的老成员;另一个是重写后的新成员。访问重名成员,访问到的将是重写后的新成员,称老成员被新成员**覆盖**了。如果希望访问被覆盖的老成员,则需要使用关键字 super 来指定,其访问形式为"**super.成员名**"。其中的关键字 super 表示超类,即所访问的成员是从超类继承来的老成员。在例 4-4 中,子类 Watch 重写了显示时间方法 show(),因此子类 Watch 中有两个重名的 show()方法。调用被覆盖的超类方法 show(),其调用形式如下:

```
super.show();                              //关键字 super 表示超类
```

例如,例 4-4 中代码第 10 行,新的 show()方法就是按这种形式来调用老的 show()方法的。

4.3.2　子类对象的定义与访问

1. 定义子类对象

与任何普通的类一样,可以使用子类来定义对象。例如,定义一个子类 Watch 的手表对象 obj:

```
Watch obj = new Watch();                    //定义一个子类 Watch 的手表对象 obj
```

计算机执行该对象定义语句,将在内存中创建一个子类 Watch 的手表对象,为其中的字段成员分配内存空间(如图 4-8 所示)。一个子类 Watch 的手表对象有 4 个字段,其中的 hour、minute、second 是从超类继承来的字段,而 band 则是子类 Watch 新添加的字段。

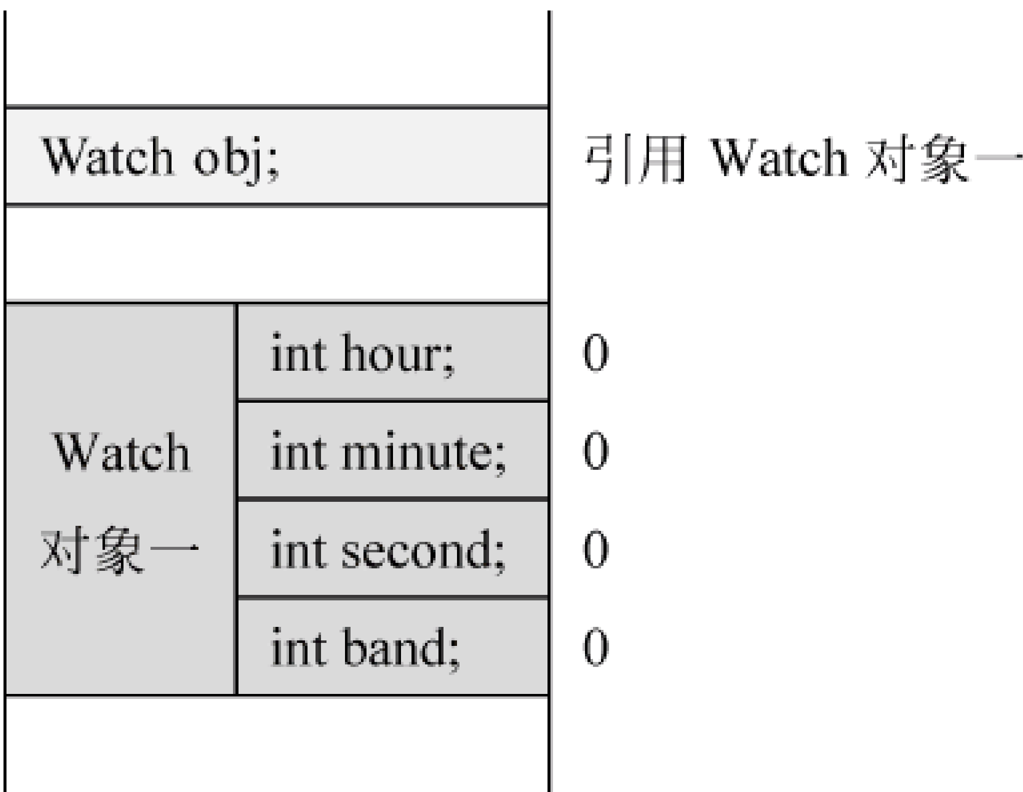


图 4-8　子类对象 obj 的内存分配

按照子类 Watch 的定义,对象 obj 将包含 8 个成员。其中 5 个是从钟表类 Clock 继承来的成员,它们分别是 obj. **hour**、obj. **minute**、obj. **second**、obj. **set**()、obj. **show**(),另外 3 个是子类 Watch 新添加的成员,它们分别是 obj. **band**、obj. **setBand**()、obj. **show**()。

2. 访问子类对象中的超类成员

子类对象中的超类成员是从超类继承来的,访问它们会有如下两点限制。

(1) 访问子类对象中的超类成员会受到其访问权限的控制。例如,以下 3 个私有的超类成员不可访问:obj. **hour**、obj. **minute**、obj. **second**。

注:超类成员的访问权限是由超类设定,并被子类原封不动继承下来的。

(2) 访问不到子类对象中被覆盖的超类成员。例如,从钟表类 Clock 继承来的显示时间方法 show()被子类 Watch 重写的新方法覆盖了。调用子类对象的方法 obj. show(),只能调用到重写后的新方法。

由于受到上述两点限制,在子类对象 obj 所包含的 5 个超类成员中,只有设置时间的方法 obj. set()可以访问,因为它是公有的,并且没有被重写。

3. 访问子类对象中的子类成员

子类成员属于子类自己定义的成员,访问它们只需要注意访问权限就可以了。例如,在子类对象 obj 所包含的 3 个子类成员中,公有成员 obj. setBand()、obj. show()可以访问,私有成员 obj. band 不可以访问。

4.3.3　保护权限

Java 语言中,类成员的访问权限总共有 4 种。第 3 章已介绍过公有权限(public)、私有权限(private)和默认权限(未指定访问权限)。本节再介绍最后一种,即**保护权限**(protected)。

表 4-1 给出了 4 种不同权限类成员的可访问范围。可以看出,具有保护权限的类成员可以在**本类**、**本包**或**子类**中访问。保护权限是在默认权限基础上,为子类定向开放的一种访问权限。

表 4-1　4 种不同的类成员访问权限

<table>
<tr><th rowspan="2">权　　限</th><th colspan="4">范　　围</th></tr>
<tr><th>在本类中访问</th><th>在本包中访问</th><th>在子类中访问</th><th>任意地方访问</th></tr>
<tr><td>私有权限(private)</td><td>可以访问</td><td colspan="3">不可访问</td></tr>
<tr><td>默认权限(未指定)</td><td colspan="2">可以访问</td><td colspan="2">不可访问</td></tr>
<tr><td>保护权限(protected)</td><td colspan="3">可以访问</td><td>不可访问</td></tr>
<tr><td>公有权限(public)</td><td colspan="4">可以访问</td></tr>
</table>

假设有程序员甲、乙、丙 3 人,程序员甲编写了一个类 A,其中定义了 3 个具有不同访问权限的字段,分别是公有字段 x、私有字段 y 和保护字段 z。下面程序员乙和丙模拟在两个不同场合使用类 A,分别访问这 3 个字段。通过对比,可以直观地了解保护权限的访问范围。

场合一:程序员乙用类 A 定义一个对象 aObj,分别访问对象 aObj 的公有字段 x、私有字段 y 和保护字段 z。

场合二:程序员丙则是用类 A 去定义一个子类 B,然后在子类 B 中访问所继承的公有字段 x、私有字段 y 和保护字段 z。

例 4-5 给出了完整的 Java 演示代码。

例 4-5　一个关于保护权限的 Java 演示程序

程序员甲:定义类 A(A.java)

```
1   public class A {
2       public   int x;              //公有权限
3       private  int y;              //私有权限
4       protected  int z;            //保护权限
5       public   void aFun() {       //在本类中访问,不受访问权限控制
6           x = 10;                  //访问公有成员,正确
7           y = 10;                  //访问私有成员,正确
8           z = 10;                  //访问保护成员,正确
9       }
10  }
```

程序员乙:使用类 A 定义对象(ATest.java)

```
1   public class ATest {   //测试类
2       public static void main(String[ ] args) {
3           //在其他类(非 A 的子类)中访问
4           A aObj = new A();   //先定义对象
5           aObj.x = 10;   //访问公有成员,正确
6           aObj.y = 10;   //访问私有成员,错误
7           //如果与类 A 不在同一包中
8           aObj.z = 10;   //访问保护成员,错误
9           //如果与类 A 在同一包中
10          aObj.z = 10;   //访问保护成员,正确
11      }
12  }  //场合一:保护权限 = 默认权限
```

程序员丙:使用类 A 定义子类 B(B.java)

```
public class B extends A {
    public   void bFun() {   //在子类 B 中访问
        x = 10;   //访问公有超类成员,正确
        y = 10;   //访问私有超类成员,错误
        z = 10;   //访问保护超类成员,正确
        //子类可以访问保护权限的超类成员
        //不管子类与超类在不在同一包中
    }
}
//场合二:保护权限 = 为子类定向开放的权限
```

通过例 4-5 可以看出，保护权限是在默认权限基础上，为**子类**定向开放的一种访问权限。

4.3.4　子类的构造方法

计算机执行定义对象语句时将创建对象，为其分配内存空间，并自动调用对象所属类的构造方法来初始化对象中的字段成员。按照来源的不同，子类中的成员可分为两种：一种是从超类继承来的**超类成员**；另一种是定义时新添加或重新定义的**子类成员**。

1. 子类构造方法的定义

为子类定义构造方法，要考虑如何初始化继承来的**超类字段**？因为它们可能是私有的，不能直接赋值。Java 语言规定，初始化超类成员必须调用超类的构造方法。调用超类构造方法的语法形式为：

```
super( 初始值列表);
```

例 4-1 定义了一个钟表类 Clock，其中包含 3 个构造方法，分别是无参构造方法、有参构造方法和拷贝构造方法。例 4-4 将这个钟表类作为超类，通过继承与扩展定义出了一个子类，即手表类 Watch。

本节以例 4-4 的手表类 Watch 为例，具体讲解如何为子类定义构造方法。例 4-6 给出了为子类 Watch 定义构造方法的示例代码。子类 Watch 的构造方法会调用超类 Clock 的构造方法。为便于阅读，这里将例 4-1 中超类 Clock 的构造方法节选出来，一并放在例 4-6 中。

例 4-6　为子类 Watch 定义的构造方法示例代码

```
1  public class Clock {                      //钟表类 Clock(超类)
2       …                                     //这里只节选例 4-1 中的构造方法,其他代码省略
3       public Clock()                        //无参构造方法:将时、分、秒字段都设为 0
4       {  hour = 0;  minute = 0;  second = 0;  }
5       public Clock(int h, int m, int s)     //有参构造方法:根据参数设置时间
6       {  hour = h;  minute = m;  second = s;  }
7       public Clock( Clock oldObj )          //拷贝构造方法:复制已有对象的时、分、秒字段
8       {  hour = oldObj.hour;minute = oldObj.minute;second = oldObj.second;  }
8  }
1  public class Watch extends Clock {        //手表类 Watch(子类)
2       …                         //这里列出为子类 Watch 定义的构造方法,其他代码省略
3       public Watch() {          //无参构造方法
4            super();             //先调用超类 Clock 的无参构造方法,初始化超类字段(时、分、秒)
5                                 //也可以调用超类 Clock 的有参构造方法:super( 0, 0, 0 );
6            band = 1;            //然后再初始化子类字段:表带类型,直接对其赋值
7       }
8       public Watch( int h, int m, int s, int b ) {    //有参构造方法:初始化时、分、秒和表带类型
9            super( h, m, s );    //需调用超类 Clock 的有参构造方法,初始化超类字段(时、分、秒)
10           band = b;            //然后再初始化子类字段:表带类型,直接对其赋值
11      }
12      public Watch( Watch oldObj ) {    //拷贝构造方法
13           super( oldObj );             //先调用超类 Clock 的拷贝构造方法,初始化超类字段
14           band = oldObj.band;          //然后再初始化子类字段:表带类型,直接对其赋值
15      }
16 }
```

子类构造方法的语法细则如下。

(1) 在子类构造方法中可以使用关键字 super 调用超类的构造方法,其目的是初始化超类字段。因为超类字段可能是私有的,在子类中不能访问,因此必须通过超类的构造方法才能进行初始化。

(2) 如果编写 super 调用语句,则该语句必须是构造方法的第一条语句。

(3) 如果没有编写 super 调用语句,则编译器会自动在构造方法的第一行增加如下调用语句:

```
super();          //调用超类的无参构造方法
```

2. 子类字段成员的初始化过程

子类中的字段成员可能有两种:一种是从超类继承来的超类字段;另一种是定义时自己添加的子类字段。子类可以在 3 个地方对字段成员进行初始化。

① 在构造方法的最开头编写 super 调用语句,调用超类的构造方法来初始化超类字段。

② 在添加新的子类字段时,可在定义时做初始化赋值。

③ 在构造方法的方法体中使用赋值语句进行初始化。可以在这里初始化新添加的子类字段,也可以初始化从超类继承来并且是可访问的超类字段。

创建子类对象时,上述初始化代码的执行顺序依次是①②③。例 4-7 给出一个初始化子类中字段成员的 Java 演示程序。**注**:请重点关注例 4-7 中子类 Sub 的定义代码。

例 4-7 一个初始化子类中字段成员的 Java 演示程序

超类 Sup(Sup.java)

```
1   class Sup {   //定义超类 Sup
2     public     int x;   //公有成员
3     private    int y;   //私有成员
4     protected int z;   //保护成员
5     public    Sup() {   //构造方法
6       //在构造方法中显示创建过程
7       System.out.println(
        "Sup enter: " + x + y + z);
8       x = 1;  y = 1;  z = 1;
9       System.out.println(
        "Sup exit: " + x + y + z);
10    }
11  }
```

子类 Sub(Sub.java)

```
1   class Sub extends Sup {   //定义子类 Sub
2     private int a = 2;      //新添加的成员,初始化②
3     public Sub() {          //子类的构造方法
4       super();              //初始化①
5       //在构造方法中显示创建过程
6       System.out.println("Sub enter: " + x + "?" + z + a);
7       a = 3;                //初始化③
8       x = 3;
9       // y = 3;//子类不能访问私有的超类成员
10      z = 3;
11      System.out.println("Sub exit: " + x + "?" + z + a);
12    }
13  }
```

```
1  public class FieldInitDemo {                //测试类(FieldInitDemo.java)
2      public static void main(String[] args) {  //主方法
3          Sub obj = new Sub();                //创建子类 Sub 的对象,将调用子类的构造方法
4      }
5  }
```

执行测试类 FieldInitDemo，主方法将创建一个子类 Sub 的对象 obj。创建对象 obj 时，计算机会自动调用子类 Sub 的构造方法。例 4-7 在构造方法中插入一些显示语句，将代码执行过程和各字段中的数据显示出来（如图 4-9 所示）。对照显示信息来阅读例 4-7 的代码，可以深入理解类中初始化代码的执行过程和顺序。

```
Problems  Javadoc  Declaration  Console
<terminated> FieldInitDemo [Java Application] C:\Java
Sup enter: 000
Sup exit: 111
Sub enter: 1?12
Sub exit: 3?33
```

图 4-9　例 4-7 程序的运行结果

4.3.5　关键字 final

英文 **final** 的原意是“**最终的**”或“**不可更改的**”。Java 语言中的关键字 final 基本保持了这个含义，但用在不同场合会有一些不同的理解。

1. final 局部变量

在方法中定义局部变量时加上关键字 final，表示该变量是一个**只读变量**（或直接称为常量），只能被赋值一次。只读变量的作用相当于是一个符号常量。例如：

```
final double PI = 3.14;          //定义时初始化，今后只能读，不能改（即不能再次赋值）
```

只读变量也可以先定义，再赋值，但只能赋值一次。例如：

```
final double PI;                 //先定义
PI = 3.14;                       //再赋值，今后只能读，不能再次赋值
```

如果局部变量是一个引用变量，则表示它只能固定引用一个对象，不能再引用其他对象或取消引用。例如：

```
final  Clock c = new Clock(8, 30, 15);//只读变量 c 是一个钟表类 Clock 的引用变量
c = new Clock(9, 30, 15);            //错误：不能改变引用，再引用其他钟表对象
```

2. final 字段

在类中定义字段成员时加上关键字 final，表示该字段是一个**只读字段**，只能被赋值一次。只读字段可以在定义时初始化，或在**构造方法**中初始化。例如：

```
public class A {
    public final  int a = 10;        //只读字段 a：定义时初始化，或在构造方法中初始化
    …                                //其他代码省略
}
```

使用类 A 定义对象 obj，定义后不能再修改字段 a 的值（即不能再次赋值）。例如：

```
A obj = new A();                 //定义一个 A 类对象 obj
System.out.println( obj.a );     //可以读取字段 a 的值并显示出来，显示结果 10
obj.a = 20;                      //错误：不能再修改字段 a 的值
```

3. final 方法

在类中定义方法成员时加上关键字 final，表示该方法是一个**最终方法**，子类不能再重

新定义(即重写)这个方法。定义 final 方法的语法形式如下:

```
[访问权限] final 返回值类型 方法名( 形式参数列表 )
{  …  }
```

4. final 类

在定义类时加上关键字 final,表示该类是一个**最终类**,不能被继承,即不能再用该类去扩展子类。定义 final 类的语法形式如下:

```
[访问权限] final class 类名
{  …  }
```

本节习题

1. 继承超类得到新的子类,子类中将不包括(　　)。
   A. 超类的私有成员　　B. 超类的保护成员
   C. 超类的公有成员　　D. 超类的构造方法
2. 访问定义在 public 类中的 protected 成员,下列访问形式中错误的是(　　)。
   A. 在同一文件的类中访问　　B. 在同一包的类中访问
   C. 在不同包的子类中访问　　D. 在不同包的非子类中访问
3. 已定义类 A:

```
class A {
    private   int x = 1;
    protected   int y = 2;
    public   int z = 3;
    public   int sumA() { return( x + y + z ); }
}
```

再通过继承与扩展定义子类 B:

```
class Bextends A {
    private   int b = 4;
    public   int sumB() {
        int s = 0;
        s += x;   s += y;   s += z;   s += b;
        return s;
    }
}
```

方法成员 sumB()中错误的语句是(　　)。
   A. s += x;　　B. s += y;　　C. s += z;　　D. s += a;
4. 定义上述子类 B 的对象 obj,则下列访问对象 obj 成员的语句中,错误的是(　　)。
   A. obj.z = 5;　　B. System.out.print(obj.sumA());
   C. obj.b = 5;　　D. System.out.print(obj.sumB());
5. 已定义类 A:

```
class A {
    private   int x;
    protected   int y;
    public   int z;
    public A(int p1, int p2, int p3) { x = p1;   y = p2;   z = p3; }   //构造方法
}
```

再通过继承与扩展定义子类 B：

```
class B   extends A {
    private   int b;
    //定义子类 B 的构造方法
}
```

则下列子类 B 的构造方法定义中，正确的是(　　)。

A. B(int p1, int p2, int p3, int p4) { x = p1;   y = p2;   z = p3;   b = p4; }

B. B(int p1, int p2, int p3, int p4) { A(p1, p2, p3);   b = p4; }

C. B(int p1, int p2, int p3, int p4) { super(p1, p2, p3);   b = p4; }

D. B(int p1, int p2, int p3, int p4): A(p1, p2, p3) { b = p4; }

6. 在定义字段成员时前面加关键字 final，其含义是(　　)。

A. 该字段不能被赋值　　B. 该字段不能被多次赋值

C. 不能读取该字段中的数据　　D. 不能显示该字段中的数据

7. 在定义方法成员时前面加关键字 final，其含义是(　　)。

A. 该方法不能被调用　　B. 该方法不能修改类中的字段成员

C. 子类不能重写该方法　　D. 子类不能调用该方法

8. 在定义类时前面加关键字 final，其含义是(　　)。

A. 不能用该类定义对象　　B. 该类不能被继承

C. 子类不能对该类的字段赋值　　D. 子类不能重写该类的方法

4.4　对象的替换与多态

在类的继承过程中，子类继承了超类除静态成员和构造方法之外的所有成员，具有超类的所有功能。也可以说，子类是超类这个大类下细分的小类，因此一个子类对象可以被当作超类对象使用。面向对象程序设计利用子类和超类之间的这种特殊关系，提出了对象的**替换与多态**，其目的是提高程序中**算法代码的重用性**。程序中最常见的算法代码形式是方法(即函数)。

本节继续通过例 4-1(4.1.1 节)所示的钟表类 Clock，具体讲解对象替换与多态的概念，以及如何运用对象替换与多态机制来提高算法代码的重用性。

4.4.1　算法代码的重用性

1. 什么是算法代码的重用性

程序中最常见的算法代码形式是方法。假设已定义一个处理 int 型数据的方法 fun()，下面分析这个方法的重用性。

```
void fun( int x ) {  System.out.println(x * x);  }          //计算并显示 x 的平方
```

调用方法 fun()可以处理 int 型数据,例如计算并显示 5 的平方。

```
fun( 5 );                    //正确:调用方法时,实参 5 的数据类型与形参 x 一致,都是 int 型
```

但是不能调用方法 fun()来处理 double 型数据,例如求实数 5.8 的平方。

```
fun( 5.8 );                  //错误:5.8 是 double 类型,与方法 fun 中形参 x 的 int 类型不一致
```

结论 1:Java 语言对数据类型一致性的要求比较严格,属于**强类型检查**的计算机语言。

注:C、C++也都属于强类型检查的语言,而 Python 则属于弱类型检查的语言。

因为数据类型不一致,不能重用方法 fun()的代码来处理 double 型数据。如需处理,程序员必须再编写一个处理 double 型数据的方法 fun(),尽管方法中求平方的算法代码完全一样。例如:

```
void fun( double x )  {  System.out.println(x * x);  }     //处理 double 型数据的重载方法 fun()
```

再进一步,如果已经定义了一个类 A 和一个处理 A 类对象的方法 aFun(),下面分析处理对象数据(即类类型数据)算法代码的重用性问题。

```
class A  {  …  }              //定义一个类 A
void aFun( A x )  {  …  }  //再定义一个处理 A 类对象的方法 aFun(),不必关注具体的算法
```

可以调用方法 aFun()来处理 A 类对象。例如:

```
A  aObj = new A();        //定义一个 A 类对象 aObj
aFun( aObj );             //正确:实参 aObj 的数据类型与方法 aFun()中形参 x 一致,都是 A 类型
```

假设还有一个类 B:

```
class B  {  …  }          //定义一个类 B
```

能否用方法 aFun()来处理 B 类的对象呢?例如:

```
B  bObj = new B();        //定义一个 B 类对象 bObj
aFun( bObj );             //错误:实参 bObj 的数据类型与方法 aFun()中形参 x 不一致
```

结论 2:不能调用处理 A 类对象的方法 aFun()来处理 B 类的对象数据。

在面向对象程序设计中,重用已有的处理超类对象的算法代码来处理子类对象,这是非常普遍的需求。如果子类能够与超类共用算法代码,它将极大地提高软件开发效率。为此,面向对象程序设计方法提出了对象的**替换**与**多态**。例如,如果类 B 是类 A 的子类:

```
class B extends A {  …  }  //类 B 继承类 A,是类 A 的子类
```

则可以用处理 A 类对象的方法 aFun()来处理 B 类对象,即子类 B 与超类 A 共用算法代码 aFun()。

2. 钟表类 Clock 及其处理算法举例

例 4-1 曾给出一个钟表类 Clock,这里再给出一个处理钟表类对象的方法 setGMT()。为了测试这个方法,将它放入到一个测试类 GMTTest 中(见例 4-8)。

例 4-8 一个处理钟表类 Clock 对象的方法 setGMT()及其测试类 GMTTest(GMTTest.java)

```
1  public class GMTTest {                              //测试类
2      public static void main(String[] args) {        //主方法
3          Clock cObj = new Clock();                   //创建一个钟表对象 cObj
4          //给定 GMT 时间 8:30:15,调用方法 setGMT()将其转成北京时间后再设置给 cObj
5          setGMT( cObj, 8, 30, 15);
6      }
7
8      //处理钟表类 Clock 对象的方法 setGMT():
9      //给定 GMT 时间,先将其转换成北京时间,然后再设置给钟表对象 obj
10     public static void setGMT(Clock obj, int hGMT, int mGMT, int sGMT) {
11         int h, m, s;            //先定义 3 个保存北京时间的变量
12         h = hGMT + 8;           //北京时间比 GMT 时间早 8 小时,即小时数加 8
13         m = mGMT;   s = sGMT;
14         obj.set(h, m, s);       //将转换后的北京时间设置给钟表对象 obj
15         obj.show();             //显示时间:GMT 时间 8:30:15 所对应的北京时间是 16:30:15
16 } }
```

这里,请读者区分两种不同的代码:一是例 4-1 中钟表类 Clock 的**类代码**;二是例 4-8 中处理钟表类 Clock 对象的**算法代码** setGMT()。

4.4.2 类族及其处理算法

在应用钟表类 Clock 的过程中,程序员可以通过继承与扩展的方法定义出各种各样的子类。例如,基于钟表类 Clock 可以定义出手表类 Watch、挂钟类 WallClock 等,还可以继续基于手表类 Watch 再定义出潜水表类 DivingWatch。例 4-9 给出上述 3 个子类的示意代码。

例 4-9 从钟表类 Clock 扩展出的的 3 个子类

手表类 Watch

```
1  class Watch extends Clock {   //手表类
2    public int band = 1;   //新添加表带类型
3    public void show() {   //重写 show()方法
4      if (band == 1)   //金属表带
5          System.out.print( "(金属表带)" );
6      else              //皮革表带
7        System.out.print( "(皮革表带)" );
8     super.show();
9  } }
```

挂钟类 WallClock

```
class WallClock extends Clock {   //挂钟类
  public int size = 12;    //新添加表盘尺寸
  public void show() {   //重写 show()方法
    System.out.print( "(" + size + "英寸)" );
    super.show();
} }
```

潜水表类 DivingWatch

```
1  class DivingWatch extends Watch{   //潜水表类
2    public int depth = 10;           //新添加最大深度
3      public void show() {           //重写 show()方法
4        System.out.print( "(" + depth + "米)" );
5    super.show();
6  } }
```

在例 4-9 中,手表类 Watch 和挂钟类 WallClock 是钟表类 Clock 的一级子类,而潜水表

类 DivingWatch 则是钟表类 Clock 的二级子类。

1. 类族

类的继承与扩展可以任意多级。用超类定义子类，子类可以继续作为超类去定义更下级的子类，这就是类的**多级继承与扩展**。经过多级继承与扩展后，超类及其下面的各级子类共同组成了一个具有继承关系和共同特性的类的家族，称为**类族**。类族中的子类具有共同的祖先，都继承了超类中的成员。例如钟表类与手表类、挂钟类、潜水表类共同组成一个钟表的类族，它们都具有钟表的功能，其中钟表类 Clock 是共同的祖先。图 4-10 给出了钟表类族的继承关系图。

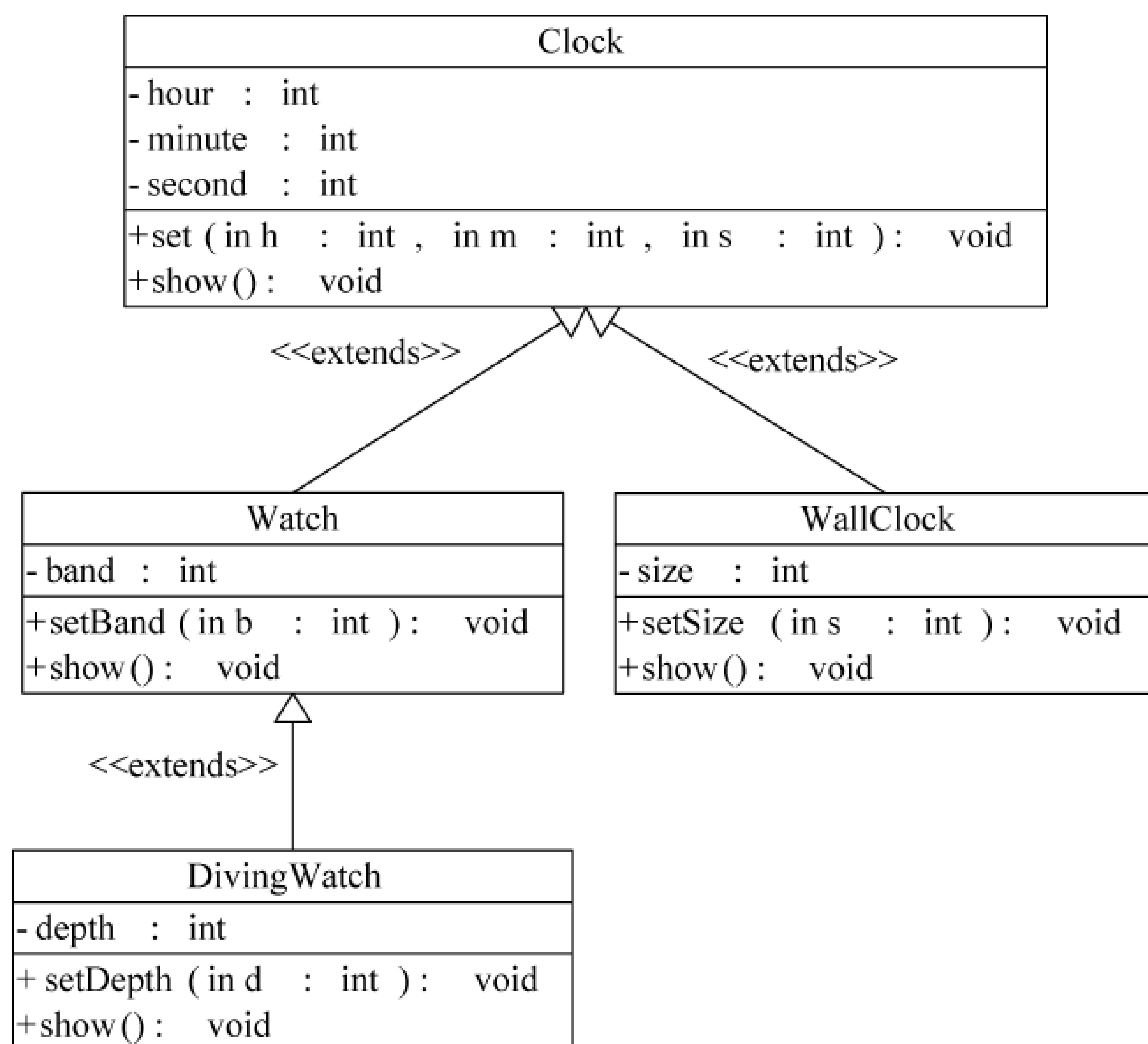


图 4-10 钟表类族的继承关系图

注意：基于钟表类 Clock，通过继承与扩展的方法定义子类，实际上是在重用钟表类 Clock 的类代码。重用超类的类代码可以有效提高子类的开发效率。

2. 同一类族中对象的多态性

在例 4-9 中，每个子类都重新定义了显示时间的方法 show()，其功能是在时间前面添加一个文字标签。下面给出超类 Clock 及其 3 个子类所显示出的不同的时间格式，其中超类所显示的时间不带文字标签，而 3 个子类则分别带有不同的文字标签。

(1) 钟表类 Clock。

```
Clock  cObj = new Clock();          //钟表类 Clock 的对象
cObj.set( 8, 30, 15 );              //将时间设置为 8:30:15
cObj.show();                        //显示时间为 8:30:15
```

(2) 手表类 Watch。

```
Watch  wObj  = new Watch();          //手表类 Watch 的对象
wObj.set( 8, 30, 15 );               //将时间设置为 8:30:15
wObj.show();                         //显示时间为(金属表带)8:30:15
```

(3) 挂钟类 WallClock。

```
WallClock  wcObj  = new WallClock();     //挂钟类 WallClock 的对象
wcObj.set( 8, 30, 15 );                  //将时间设置为 8:30:15
wcObj.show();                            //显示时间为(12 英寸)8:30:15
```

(4) 潜水表类 DivingWatch。

```
DivingWatch  dwObj  = new DivingWatch();  //潜水表类 DivingWatch 的对象
dwObj.set( 8, 30, 15 );                   //将时间设置为 8:30:15
dwObj.show();                             //显示时间为(10 米)(金属表带)8:30:15
```

可以看出,同一类族中不同钟表对象会显示出不同的时间格式,称这些钟表对象表现出了**多态性**(polymorphism)。

对象多态性这个术语是从生物多态性借鉴而来的。例如,米老鼠、唐老鸭分别是老鼠类和鸭子类的对象。下达指令 **Go**,米老鼠将迈开 4 条腿迅速移动,唐老鸭则是迈开两条腿蹒跚而行。不同生物对象在执行相同指令 Go 的时候会表现出不同的形态,这就是生物对象的多态性。

在面向对象程序设计中,不同对象可能具有同名的方法成员。例如,使用类 A、类 B 分别定义对象 aObj 和 bObj,假设它们都有一个名为 fun()的方法成员,但算法和功能各不相同。将调用对象方法成员 fun()的操作做如下类比。

- 调用对象的方法成员 fun()。例如:

  ```
  aObj.fun();  bObj.fun();
  ```

  这类似于向对象 aObj、bObj 分别下达了相同的指令 fun。
- 行方法 fun()。这类似于对象 aObj、bObj 各自执行指令 fun。
- 不同的 fun()方法完成不同的功能。这类似于对象 aObj、bObj 表现出不同的形态。

面向对象程序设计借用拟人化的说法,将调用对象的某个方法成员称为向对象发送一条**消息**,将执行方法成员完成某种程序功能称为对象**响应**该消息。不同对象接收相同的消息,但会表现出不同的行为,这就称为对象的多态性,或称对象具有多态性。

从程序角度看,**对象多态性**就是调用不同对象的同名方法成员,但所执行的方法不同,完成的程序功能也不同。面向对象程序设计只关注**子类**与**超类**之间存在的多态性问题。

3. 子类与超类共用算法代码

例 4-8 中编写了一个处理钟表类 Clock 对象的方法 setGMT(),其功能是给定 GMT 时间,先将其转换成北京时间,然后再设置给钟表对象。为便于阅读,将这个方法的定义代码完整地摘录过来,其内容如下:

```
public static  void setGMT(Clock obj, int hGMT, int mGMT, int sGMT) {
    int h, m, s;              //先定义 3 个保存北京时间的变量
    h = hGMT + 8;             //北京时间比 GMT 时间早 8 小时,即小时数加 8
    m = mGMT;  s = sGMT;
    obj.set(h, m, s);         //将转换后的北京时间设置给钟表对象
    obj.show();               //显示转换后的北京时间
}
```

调用方法 setGMT()可以为 Clock 类的钟表对象设置 GMT 时间。例如：

```
Clock cObj = new Clock();      //创建一个 Clock 类的钟表对象 cObj
setGMT( cObj, 8, 30, 15);      //传递一个 GMT 时间 8:30:15
```

方法 setGMT()接收 GMT 时间 8:30:15,将其转成北京时间 16:30:15,然后再设置给钟表对象 cObj。方法 setGMT()在设置时间后会调用钟表对象的显示时间方法 show(),显示所设定的北京时间。例如,执行上述调用方法 setGMT()的语句会显示出如下格式的时间：

```
16:30:15
```

注：这个格式是超类 Clock 所显示出的时间格式。

问题：能否重用处理超类 Clock 对象的方法 setGMT(),为子类对象设置 GMT 时间?例如,为 Watch 类的手表对象设置 GMT 时间：

```
Watch wObj = new Watch();      //创建一个 Watch 类的手表对象 wObj
setGMT( wObj, 8, 30, 15);      //传递一个 GMT 时间 8:30:15
```

手表类 Watch 是钟表类 Clock 的子类。如果能够重用处理超类 Clock 对象的方法 setGMT(),继续为子类 Watch 对象设置 GMT 时间,那就意味着子类可以与超类共用**算法代码**。

Java 语言是强类型检查的计算机语言。调用方法时,实参的数据类型原则上应当与形参一致。仔细分析一下方法 setGMT()的定义代码,会发现有如下两个问题需要讨论。

讨论 1：形参 obj 是一个超类 Clock 的引用变量,实参 wObj 是一个子类 Watch 的引用变量。形实结合时,实参和形参的类型不一致怎么办?

讨论 2：通过超类 Clock 的引用变量 obj 调用子类 Watch 对象的显示时间方法 show(),会调用哪个 show()?请注意,钟表类族中的每个子类都继承了超类的显示时间方法 show(),同时又重新定义了自己新的显示时间方法 show(),即子类中有多个同名的方法 show()。

针对这两个问题,Java 语言专门制定了对象替换与多态的语法规则。

4.4.3 对象的替换与多态

为了让类族能够共用算法代码,Java 语言为超类引用变量**引用**和**访问**子类对象专门制定了相关的语法规则。

1. 超类引用变量引用子类对象

Java 语言为超类引用变量**引用**子类对象专门制定了**对象替换**语法规则：可以将子类对象的引用赋值给超类的引用变量,即超类引用变量可以引用子类对象。例如：

```
Clock cObj1 = new Watch();            //类 Watch 是类 Clock 的一级子类
Clock cObj2 = new WallClock();        //类 WallClock 是类 Clock 的一级子类
Clock cObj3 = new DivingWatch();      //类 DivingWatch 是类 Clock 的二级子类
```

对象替换，实际上是将子类对象当作一个超类对象来使用。例如，可以将一个潜水表(子类对象)当作一个普通钟表(超类对象)来使用。

反过来，也可以将超类的引用变量赋值给子类的引用变量。需要注意的是，将超类的引用变量赋值给子类的引用变量，赋值时必须进行**强制类型转换**。例如：

```
Watch wObj = ( Watch) cObj1;             //cObj1 必须确实引用了一个类 Watch 的对象
WallClock wcObj = ( WallClock) cObj2;    //cObj2 必须确实引用了一个类 WallClock 的对象
DivingWatch dwObj = ( DivingWatch) cObj3;//cObj3 必须确实引用了一个类 DivingWatch 的对象
```

将超类的引用赋值给子类引用变量，其含义是：将超类引用变量所引用的子类对象赋值给该子类的另一个引用变量。

2. 通过超类引用变量访问子类对象的成员

超类引用变量在引用某个子类对象后，可以通过该引用变量访问子类对象的成员。按照来源的不同，子类对象中的成员可分为两种：一是从超类继承来的成员，即**超类成员**(或称**老成员**)；二是定义子类时新添加或重新定义的**新成员**，即**子类成员**。

Java 语言为超类引用变量访问子类对象成员专门制定了如下两条**对象多态**语法规则。

(1) 通过超类引用变量访问子类对象的成员，只能访问从超类继承来的老成员，不能访问子类新添加的新成员。例如：

```
Clock cObj = new Watch();     //超类引用变量 cObj 引用了一个子类 Watch 的手表对象
cObj.set( 8, 30, 15 );        //正确:可以访问子类对象中的 set(),它是从超类继承来的老成员
cObj.band = 1;                //错误:不能访问子类对象中的 band,它是子类添加的新成员
```

(2) 如果子类重新定义了超类成员，这时子类中将有两个同名的成员：一个是从超类继承来的老成员；另一个是子类重写的新成员。通过超类引用变量访问子类对象中的这个同名成员，将自动调用子类重写的新成员。例如，子类 Watch 重新定义了显示时间方法 show()，其功能是在时间前面添加一个描述表带类型的文字标签。

```
Clock cObj = new Watch();  //超类引用变量 cObj 引用了一个子类 Watch 的手表对象
cObj.set( 8, 30, 15 );     //设置手表对象的时间
cObj.show();          //自动调用子类 Watch 重写的新方法 show(),显示结果为(金属表带)8:30:15
```

需要说明的是，类中的静态成员(static)**不遵循**上述对象多态语法规则。通过超类引用变量访问子类对象中重写的静态成员，访问到的将是从超类继承来的老成员，而不是子类重写的新成员。

3. 类族共用算法代码

Java 语言通过对象的替换和多态机制，巧妙实现了同一类族可以共用算法代码。例如，钟表类族可以共用设置 GMT 时间的算法代码 setGMT()。

(1) 钟表类 Clock(超类)。

```
Clock  cObj = new Clock();      //钟表类 Clock 的对象
setGMT( cObj, 8, 30, 15);       //为超类 Clock 的钟表对象设置 GMT 时间,显示结果为 16:30:15
```

(2) 手表类 Watch(一级子类)。

```
Watch  wObj = new Watch();      //手表类 Watch 的对象
setGMT( wObj, 8, 30, 15);       //为子类手表对象设置 GMT 时间,显示结果为(金属表带)16:30:15
```

(3) 挂钟类 WallClock(一级子类)。

```
WallClock  wcObj = new WallClock();     //挂钟类 WallClock 的对象
setGMT( wcObj, 8, 30, 15);        //为子类挂钟对象设置 GMT 时间,显示结果为(12 英寸)16:30:15
```

(4) 潜水表类 DivingWatch(二级子类)。

```
DivingWatch  dwObj = new DivingWatch();//潜水表类 DivingWatch 的对象
setGMT( dwObj, 8, 30, 15); //为潜水表对象设置 GMT 时间,显示结果为(10 米)(金属表带)16:30:15
```

从各钟表对象所显示的时间格式来看,类族不仅可以共用算法代码,而且共用时还能继续保持对象的多态性。Java 语言的对象替换和多态语法规则是实现上述功能的关键。

面向对象程序设计通过继承与扩展实现了对类代码的重用,通过对象替换和多态又实现了对算法代码的重用。例如,通过继承与扩展可以重用已有钟表类 Clock 的类代码来定义子类,然后通过对象替换与多态又让子类能够继续重用处理 Clock 类的算法代码 setGMT()。综合运用这两项技术可以充分重用已有的程序代码,有效提高新程序的开发效率。

4. 运算符 instanceof

一个超类引用变量所引用的可能是超类对象,也可能是子类对象。如何确定引用变量到底引用的是哪个类的对象呢?Java 语言提供了一个特殊的运算符 **instanceof**,其功能是检查引用变量是否引用了某个类的对象。instanceof 的使用语法如下:

```
引用变量名 instanceof 类名
```

运算符 instanceof 的计算结果是 boolean 类型。例如:

```
Clock cObj = new Watch();            //引用变量 cObj 引用了一个子类 Watch 的对象
```

则表达式"**cObj** instanceof **Watch**"的计算结果为 **true**,cObj 确实引用了一个 Watch 类的对象;表达式"**cObj** instanceof **Clock**"的计算结果也为 **true**,子类被认为是一种超类;表达式"**cObj** instanceof **DivingWatch**"的计算结果则为 **false**,超类不是一种子类。

子类与超类之间的关系如下。

(1) **子类可被认为是一种超类**。例如,手表是一种钟表,手表类是钟表这个大类下的一个细分小类。**注**:反过来不成立,例如钟表不能被认为是一种手表。

(2) **超类可以代表子类**。例如,钟表是手表、挂钟和潜水表等经泛化、抽象后得到的上层概念,可代表各种不同形式的钟表。

本节习题

1. 对象多态性是程序中的某种现象，这种现象是（　　）。
   A. 显示同一对象的不同字段成员，会得到不同的显示结果
   B. 调用同一对象的不同方法成员，会得到不同的处理结果
   C. 显示不同对象的同名字段成员，会得到不同的显示结果
   D. 调用不同对象的同名方法成员，会得到不同的处理结果
2. Java语言重点关注的对象多态性形式是（　　）。
   A. 同类多个对象之间的多态
   B. 同一类族不同对象之间的多态
   C. 不同组合类对象之间的多态
   D. 组合类对象和包装类对象之间的多态
3. 下列关于对象替换与多态的描述中，错误的是（　　）。
   A. 对象替换与多态的目的是提高程序中算法代码的重用性
   B. 对象替换与多态的基础是子类与超类之间具有相似性
   C. 通过类的继承与扩展可以实现类代码的重用
   D. 通过对象替换与多态可以实现类代码的重用
4. 下列关于对象替换语法规则的描述中，错误的是（　　）。
   A. 可以将子类对象的引用赋值给超类的引用变量
   B. 超类的引用变量可以引用子类对象
   C. 可以将超类的引用变量直接赋值给子类的引用变量
   D. 可以将超类的引用变量赋值给子类的引用变量，赋值时必须进行强制类型转换
5. 下列关于对象多态语法规则的描述中，错误的是（　　）。
   A. 通过超类引用变量访问子类对象的成员，只能访问其中超类定义过的成员
   B. 通过超类引用变量访问子类对象的成员，不能访问其中新添加的成员
   C. 如果子类重写了超类成员，通过超类引用变量所访问到的是重写前的老成员
   D. 如果子类重写了超类成员，通过超类引用变量所访问到的是重写后的新成员
6. 定义如下的超类A和子类B：

```
class A {
    public void fun() { … }        //代码省略
}
class B  extends A {
    public void fun() { … }        //重写 fun(),代码省略
}
```

按如下形式创建一个子类B的对象，然后调用其方法成员fun()：

```
B b = new B();  b.fun();
```

上述调用方法成员fun()的执行过程是（　　）。

A. 执行类 A 定义的 fun()
B. 先执行类 A 定义的 fun(),再执行类 B 重写的 fun()
C. 执行类 B 重写的 fun()
D. 先执行类 B 重写的 fun(),再执行类 A 定义的 fun()

7. 使用第 6 题中的超类 A 和子类 B,按如下形式创建一个子类 B 的对象,然后调用其方法成员 fun():

```
A a = new B();  a.fun();
```

上述调用方法成员 fun()的执行过程是(　　)。
A. 执行类 A 定义的 fun()
B. 先执行类 A 定义的 fun(),再执行类 B 重写的 fun()
C. 执行类 B 重写的 fun()
D. 先执行类 B 重写的 fun(),再执行类 A 定义的 fun()

8. 定义如下的超类 A 和子类 B:

```
class A {
    public void fun() { … }                //代码省略
}
class B  extends A {
    public void fun() { … }                //重写 fun(),代码省略
    public void fun1() { … }               //新添加 fun1(),代码省略
}
```

按如下形式创建两个子类 B 的对象,然后分别访问其下级成员:

```
A a = new B();  B b = new B();             //创建对象
a.fun();  a.fun1();  b.fun();  b.fun1(); //访问对象的下级成员
```

上述访问对象下级成员的语句中,错误的是(　　)。
A. a.fun();　　B. a.fun1();　　C. b.fun();　　D. b.fun1();

4.5 抽象类与接口

程序员通过继承与扩展可以重用已有超类的代码,这样就能站在更高的起点上开发程序,提高开发效率,降低开发难度。程序员也可以合理运用继承与扩展来凝练类代码,有效减少程序中的重复代码。

4.5.1 凝练类代码

使用面向对象程序设计方法来设计解决某个实际问题的计算机程序,先从实际问题中提取出一个个具体的客观对象,并将具有共性的对象划分成类。可以继续将多个不同类中的共性抽象出来形成超类,编码时再从超类继承,扩展出各个不同的类。

例 4-10 给出一个本科生类和研究生类的 Java 示意代码。

例 4-10　一个本科生类和研究生类的 Java 示意代码

```
Undergraduate: 本科生类
1  public class Undergraduate {     //本科生类
2    public char Name[], ID[];     //姓名、学号
3    public int Age;               //年龄
4    public float Score;           //课堂成绩
5    public float DesignScore; //毕业设计成绩
6
7    public void Input() { … } //输入学生信息
8    public void ShowInfo() { … } //显示学生信息
9    public float TotalScore() { … } //计算总成绩
10 }
```

```
Graduate: 研究生类
public class Graduate {        //研究生类
  public char Name[], ID[]; //姓名、学号
  public int Age;             //年龄
  public float Score;         //课堂成绩
  public float PaperScore;    //毕业论文成绩
  public int Thesis;          //发表论文数量
  public void Input() { … }   //输入学生信息
  public void ShowInfo() { … } //显示学生信息
  public float TotalScore() { … } //计算总成绩
}
```

例 4-10 中的本科生类和研究生类中有部分重复代码。例如,姓名、学号、年龄、课堂成绩等字段是一样的,在输入和显示学生信息方法中关于基本信息部分的代码也是重复的。可以将这两个类中的共性部分抽象出来形成一个学生类,然后将学生类作为超类,通过继承与扩展的方法再还原出本科生类和研究生类(见例 4-11)。

例 4-11　抽象超类(学生类)后再扩展本科生类和研究生类的 Java 示意代码

```
   Student: 学生类(抽象出的超类)
1  public class Student {             //超类: 学生类
2    public char Name[], ID[];       //姓名、学号
3    public int Age;                 //年龄
4    public float Score;             //课堂成绩
5    public void Input() { … }       //输入学生基本信息
6    public void ShowInfo() { … }    //显示学生基本信息
7  }
```

```
   Undergraduate: 本科生类(子类)
1  public class Undergraduate extends Student {
2    public float PracticeScore;//毕业设计成绩
3
4    public float TotalScore() { … }//计算总成绩
5    public void Input() { … }//重写输入方法
6    public void ShowInfo() { … }//重写显示方法
7  }
```

```
Graduate: 研究生类(子类)
class Graduate extends Student {
  public double PaperScore;//毕业论文成绩
  public int Thesis;//发表论文数量
  public float TotalScore() { … }//计算总成绩
  public void Input() { … }//重写输入方法
  public void ShowInfo() { … }//重写显示方法
}
```

例 4-11 所定义出的本科生类、研究生类在功能上与例 4-10 完全一样,但例 4-11 通过抽象超类有效减少了程序中的重复代码。

面向对象程序设计方法从对象抽象出类,从类再继续抽象出超类,这是一个“自底向上,逐步抽象”的过程。越往上,类就越宽泛、越抽象。

4.5.2　抽象方法与抽象类

例 4-12 给出一个圆形类和长方形类的 Java 示意代码。

例 4-12 一个圆形类和长方形类的 Java 示意代码

```
   Circle: 圆形类                                      Rectangle: 长方形类
1  public class Circle {              //圆形类        public class Rectangle {          //长方形类
2    publicdouble r;                  //半径            publicdouble a, b;              //长、宽
3    publicdouble area() { … }        //求面积          publicdouble area() { … }       //求面积
4    publicdouble len() { … }         //求周长          publicdouble len() { … }        //求周长
5  }                                                  }
```

求面积、周长是圆形类和长方形类的共性部分，可以再抽象出一个如下的几何形状类 Shape：

```
public abstract class Shape {          //形状类
    public abstract double area();     //求面积方法的签名
    public abstract double len();      //求周长方法的签名
}
```

几何形状类 Shape 有两个方法成员，分别是求面积方法 area()和求周长方法 len()。仔细分析，这两个方法成员无法定义，因为形状是一个纯抽象的概念，无法计算其面积和周长。

1. 抽象方法与抽象类

Java 语言将只给出方法签名，但没有方法体的方法称为**抽象**(abstract)**方法**，定义时需使用关键字 **abstract** 进行修饰；将含有抽象方法的类称为**抽象类**，定义时也必须使用关键字 abstract。

抽象类的语法细则如下。

(1) 抽象类不能实例化。

不能使用抽象类创建对象(即不能实例化)，因为抽象类中含有未定义的抽象方法，其类型定义还不完整。

(2) 抽象类可以作为超类定义子类。

- 抽象类可以作为超类定义子类。子类将继承抽象类中除静态成员和构造方法之外的所有成员，包括其中的抽象方法。
- 抽象方法只设计了方法的调用接口，即只定义了方法签名，但没有提供方法体代码。子类继承了抽象方法的调用接口，还需要为抽象方法编写具体的算法代码，这被称为是**实现**(implement)抽象方法。
- 子类如果实现了所有的抽象方法，那么它就变成了一个普通的类，可以实例化；否则子类仍然是一个抽象类，定义时继续使用关键字 abstract。
- 可以定义抽象类的引用变量，所定义的引用变量可以引用其子类的实例化对象。

(3) 抽象类可以包含字段和非抽象的方法。

抽象类可以包含抽象方法，也可以包含字段和非抽象的方法。一个类只要包含一个抽象方法，则这个类就是抽象类，定义时必须使用关键字 abstract 进行修饰。

本节前面给出的几何形状类 Shape 就是一个抽象类，其中包含两个抽象方法 area()和 len()。下面的例 4-13 给出一个继承几何形状类 Shape 并实现其中抽象方法的圆形类和长

方形类的 Java 示例代码。

例 4-13 继承几何形状类 Shape 并实现其中抽象方法的圆形类和长方形类的 Java 示例代码

```
   Circle: 圆形类
1  public class Circle extends Shape {//圆形类
2    public double r;           //添加字段半径
3    publicdouble area()        //实现抽象方法
4    {  return ( 3.14 * r * r );  }
5    publicdouble len()         //实现抽象方法
6    {  return ( 3.14 * 2 * r );  }
7    public Circle(double x) //构造方法
8    {  r = x;  }
9  }

   Rectangle: 长方形类
   public class Rectangleextends Shape {//长方形类
     public double a, b;        //添加字段长、宽
     publicdouble area()        //实现抽象方法
     {  return ( a * b );  }
     publicdouble len()         //实现抽象方法
     {  return ( 2 * (a + b) );  }
     public Rectangle(double x, double y) //构造方法
     {  a = x;  b = y;  }
   }
```

例 4-13 中的圆形类 Circle 和长方形类 Rectangle 都是抽象类 Shape 的子类，它们共同组成了一个以抽象类 Shape 为超类的类族，可称为几何形状类族。

2. 抽象类的应用

抽象类及其子类共同组成一个类族。应用抽象类主要有以下两个方面的考虑：一是统一类族对外的使用接口；二是让类族共用算法代码。

1）统一类族对外的使用接口

通常，子类继承超类是为了重用超类的代码。如果超类是抽象类，其中的抽象方法并没有定义方法体的算法代码。超类定义抽象方法不是为了重用其代码，而是为了统一类族对外的使用接口。

在超类中定义抽象方法，子类按照各自的功能要求实现这些抽象方法，这样类族中的所有子类都具有相同的使用接口。例如：

```
Circle   cObj  =  new Circle( 10 );                       //定义一个圆形类对象 cObj
Rectangle  rObj  =  new Rectangle( 5, 10 );               //定义一个长方形类对象 rObj
System.out.println( cObj.area()  + ", "  + cObj.len() );  //显示圆形对象的面积和周长
System.out.println( rObj.area()  + ", "  + rObj.len() );  //显示长方形对象的面积和周长
```

可以看出，不管是圆形还是长方形，只要是几何形状类族中的类，它们计算面积的接口都被统一成 **area()**，计算周长的接口则被统一成 **len()**。统一之后的接口便于记忆，便于使用。

2）类族共用算法代码

抽象类中的抽象方法也具有多态性。定义抽象方法的另一个目的是在统一类族接口之后利用对象多态性，让类族共用算法代码。例如，抽象类 Shape 类族中的所有子类都可以共用如下的 shapeInfo()方法来显示面积、周长等形状信息。

```
publicvoid shapeInfo( Shape sObj ) {                       //显示面积、周长等形状信息
    System.out.println( sObj.area()  + ", "  + sObj.len() );
}
```

调用这个 shapeInfo()方法可以显示圆形对象的面积和周长，也可以显示长方形对象的

面积和周长。例如：

```
Circle  cObj = new Circle( 10 );          //定义一个圆形类对象 cObj
Rectangle  rObj = new Rectangle( 5, 10 ); //定义一个长方形类对象 rObj
shapeInfo( cObj );                        //共用方法 shapeInfo,显示圆形对象的形状信息
shapeInfo( rObj );                        //共用方法 shapeInfo,显示长方形形对象的形状信息
```

抽象类 Shape 类族中的子类之所以能够共用 shapeInfo()方法，这是因为它们属于同一个类族，具有统一的计算面积和周长的接口。

4.5.3 接口

接口(interface)是一种特殊的抽象类，其中只包含抽象方法、静态方法或静态只读字段等特殊成员。例如 4.5.2 节中的几何形状类 Shape，其中定义了两个计算面积和周长的方法 area()和 len()，但它们没有给出实现具体算法的方法体，只给出了方法签名(即调用接口)。因此，几何形状类 Shape 只能说是一种可以计算面积和周长的接口。

Java 语言将接口从抽象类中独立出来，单独作为一种引用数据类型。接口在 Java 语言中占有非常重要的地位。

1. 接口的定义与实现

Java 语法：定义接口和实现接口

```
[public] interface 接口名  [ extends 父接口名列表] {
    [public static final] 数据类型 公有静态只读字段名 = 初始值;
    …
    [public static] 返回值类型 公有静态方法名( 形式参数列表)  {  …  }
    …
    [public abstract] 返回值类型 公有抽象方法名( 形式参数列表);
    …
}

class 类名 implements 接口名列表 {
    实现从接口继承的抽象方法
    定义类的其他成员
}
```

语法说明：

- 定义接口时使用关键字 **interface**。
- **接口名**需符合标识符的命名规则，习惯上以大写字母开头；通常也会以“-able”或“-ible”结尾，表示可做什么操作的接口。
- 定义接口时可以使用关键字 **extends** 继承其他接口(称为父接口)，然后在此基础上进行扩展。接口可以继承多个父接口，多个父接口之间用逗号隔开。接口可以多继承。
- 接口的成员只能有 3 种，分别是**公有静态只读字段**(public static final，可省略)、**公有静态方法**(public static，可省略)和**公有抽象方法**(public abstract，可省略)，它们都

是公有权限。接口中的抽象方法只有方法签名，但没有方法体，其目的是向外界提供一种统一的操作接口标准。某些接口仅包含一个抽象方法成员，称为**功能接口**(functional interface)。某些接口不包含任何成员，是一个空接口，称为**标记接口**(marker interface)。

- 接口可以被类**实现**。定义类时可使用关键字 **implements** 实现某个接口，然后在类中对接口里的抽象方法进行重新定义，并编写完整的方法体代码，这个过程就被称为是类对接口的实现。类实现接口的目的是继承其中的接口设计，然后按照统一的接口标准(即方法具有统一的调用接口)向外界提供服务(即方法可以被外界调用)。实现了某个接口的类被称为该接口的子类。
- 一个类只能继承一个超类(即**单继承**)，但可以实现多个接口(即**多实现**)。实现多个接口时，多个接口之间用逗号隔开。
- 接口是一种特殊的**引用数据类型**。可以用接口定义引用变量，但不能创建对象。接口类型的引用变量可以引用其子类的对象。

2. 一个儿童手表类举例

这里以一个儿童手表类 ChildrenWatch 为例，具体讲解接口的定义和实现方法。儿童手表首先是一个手表，然后又增加了打电话和定位的功能。

可以预先为打电话功能定义一个统一的操作接口标准，然后让儿童手表类实现该接口，并按接口标准实现具体的打电话功能。同理，也可以为定位功能定义一个统一的操作接口标准，然后让儿童手表类实现具体的定位功能。

(1) 定义一个**可以打电话的**接口 Callable。

```
public interface Callable {
    int pNumber = 1234;         //本机号码：公有静态只读字段，本质上就是一个常量
    void call(int number);      //打电话的操作接口标准：公有抽象方法
    void answer();              //接电话的操作接口标准：公有抽象方法
}
```

(2) 定义一个**可以定位的**接口 Positionable。

```
public interface Positionable {
    void showPosition();        //显示定位的操作接口标准：公有抽象方法
}
```

(3) 定义一个**儿童手表类** ChildrenWatch。

定义儿童手表类 ChildrenWatch，可以继承 4.3.1 节例 4-4 的手表类 Watch，然后再实现可以打电话的接口 Callable 和可以定位的接口 Positionable。例 4-14 给出了一个完整的儿童手表类 ChildrenWatch 定义及测试代码。

例 4-14　一个儿童手表类 ChildrenWatch 定义(ChildrenWatch.java)及测试代码

```
1  public class ChildrenWatch extends Watch implements Callable, Positionable {
2      //继承手表类 Watch，同时实现接口 Callable 和 Positionable
3      public void call(int number)                    //实现接口 Callable 所规定的打电话方法
4      {  System.out.println("Call " + number);  }    //显示一个模拟打电话的提示信息
```

```
5       public void answer()                    //实现接口 Callable 所规定的接电话方法
6       {  System.out.println("Answer a call");  } //显示一个模拟接电话的提示信息
7
8       public void showPosition()             //实现接口 Positionable 所规定的显示定位方法
9       {  System.out.println("Show position");  }    //显示一个模拟定位的提示信息
10  }
```

```
1  public class CWatchTest {                          //测试类(CWatchTest.java)
2      public static void main(String[] args) {       //主方法
3          ChildrenWatch cw = new ChildrenWatch();  //使用儿童手表类定义对象
4          cw.set(8, 30, 15);                         //设置手表时间
5          cw.show();                                 //显示手表信息:(金属表带)8:30:15
6          cw.call(6789);                             //打电话.模拟显示结果: Call 6789
7          cw.answer();                               //接电话.模拟显示结果: Answer a call
8          cw.showPosition();                         //显示定位.模拟显示结果: Show position
9      }
10  }
```

例4-14中的儿童手表类ChildrenWatch继承手表类Watch,这样就具备了手表的功能;然后又实现可以打电话的接口Callable,即遵照方法call()和answer()的接口标准编写具体的打电话算法代码,这样就定义出了一个可以打电话的手表类;再实现可以定位的接口Positionable,即遵照方法showPosition()的接口标准编写具体的定位算法代码,最终定义出一个可以打电话的和可以定位的手表类,即儿童手表类。

测试类CWatchTest中的主方法使用类ChildrenWatch定义儿童手表对象。在使用儿童手表对象的打电话功能时,同样也是遵照接口Callable所规定的接口标准来调用方法call()和answer();而在使用儿童手表对象的定位功能时,遵照的则是接口Positionable中方法showPosition()的接口标准。

假设甲是定义儿童手表类ChildrenWatch的程序员,乙是使用儿童手表类ChildrenWatch的程序员。接口Callable的作用是为甲、乙两位程序员提供一个关于打电话功能的接口设计,不管是实现功能或是使用功能都应遵守这个接口设计。接口Positionable的作用与此类似,它为两位程序员制定了一个关于定位功能的接口设计。

总结一下,接口的作用是在**定义类**的程序员和**使用类**的程序员之间约定一个双方应当共同遵守的**方法签名**(**即调用接口**)设计。在约定好接口设计之后,定义类的程序员和使用类的程序员各自编写自己的程序代码,并行开发。

3. 抽象方法的默认方法体

定义接口时可以为其中的抽象方法提供一个默认方法体,定义时使用关键字 **default**。例如:

```
public interface Positionable {                    //定义一个可以定位的接口
    //void showPosition() ;                        //无默认方法
    default void showPosition()                    //有默认方法
    {  System.out.println( "Show position" );  }   //默认方法体
}
```

实现接口 Positionable 时,类可以重新定义抽象方法 showPosition(),编写具体的方法体代码;或者不重新定义,这时类将直接使用接口里的默认方法体。

4. 继承超类与实现接口

1) 子类继承超类

子类是超类的扩展,但仍属于超类,与超类具有类属关系。超类的功能是子类的主要功能。例如,儿童手表继承手表,它仍然是一种手表,手表是儿童手表的主要功能。

2) 子类实现接口

子类实现接口,为接口实现具体功能,但这些功能只是子类的次要功能。例如,实现接口 Callable 后,儿童手表就变成了一种可以打电话的手表;再实现接口 Positionable 后,儿童手表就变成了可以打电话的和可以定位的手表。子类与接口之间没有类属关系。

3) 类只能单继承,接口可以多实现

一个类只能继承一个超类,但可以实现多个接口。

5. 接口的应用

1) 使用接口定义引用变量

接口是一种引用数据类型,可以用来定义引用变量,但不能创建对象,即不能实例化。接口类型的引用变量可以引用实现本接口的子类对象,但访问对象时只能访问到该接口曾经定义过的成员。**注**:这一点与通过超类引用变量访问子类对象是一样的。例如:

```
ChildrenWatch cw = new ChildrenWatch();//定义一个儿童手表类的对象 cw
//通过超类引用变量访问子类对象
Watch r1 = cw;                    //超类 Watch 的引用变量 r1,引用子类对象 cw
r1.show();                        //通过 r1 只能访问超类 Watch 曾经定义过的成员
//通过 Callable 接口引用变量访问子类对象
Callable r2 = cw;                 //接口 Callable 的引用变量 r2,引用子类对象 cw
r2.call(6789);  r2.answer();      //通过 r2 只能访问接口 Callable 曾经定义过的成员
//通过 Positionable 接口引用变量访问子类对象
Positionable r3 = cw;             //接口 Positionable 的引用变量 r3,引用子类对象 cw
r3.showPosition();                //通过 r3 只能访问接口 Positionable 曾经定义过的成员
```

2) 让同一接口族共用算法代码

一个接口可以被多个类实现,这些类就构成一个具有统一接口的类族,称为**接口族**。和类族一样,同一接口族中的类可以利用对象替换与多态机制共用算法代码。图 4-11 给出一个类族和接口族的关系示意图。图 4-11 中有 3 个类族 A、B、C,还有两个接口族,即接口族 1 和接口族 2。

通过图 4-11 将类族与接口族做一个对比。

(1) 类族。

类族中,各个类之间有紧密的类属关系。利用对象替换与多态机制,同一类族中的超类和各子类之间可以共用算法代码。例如,图 4-11 所示类族 A 中的超类和各子类之间可以共用算法代码。同理,类族 B、类族 C 也是这样。

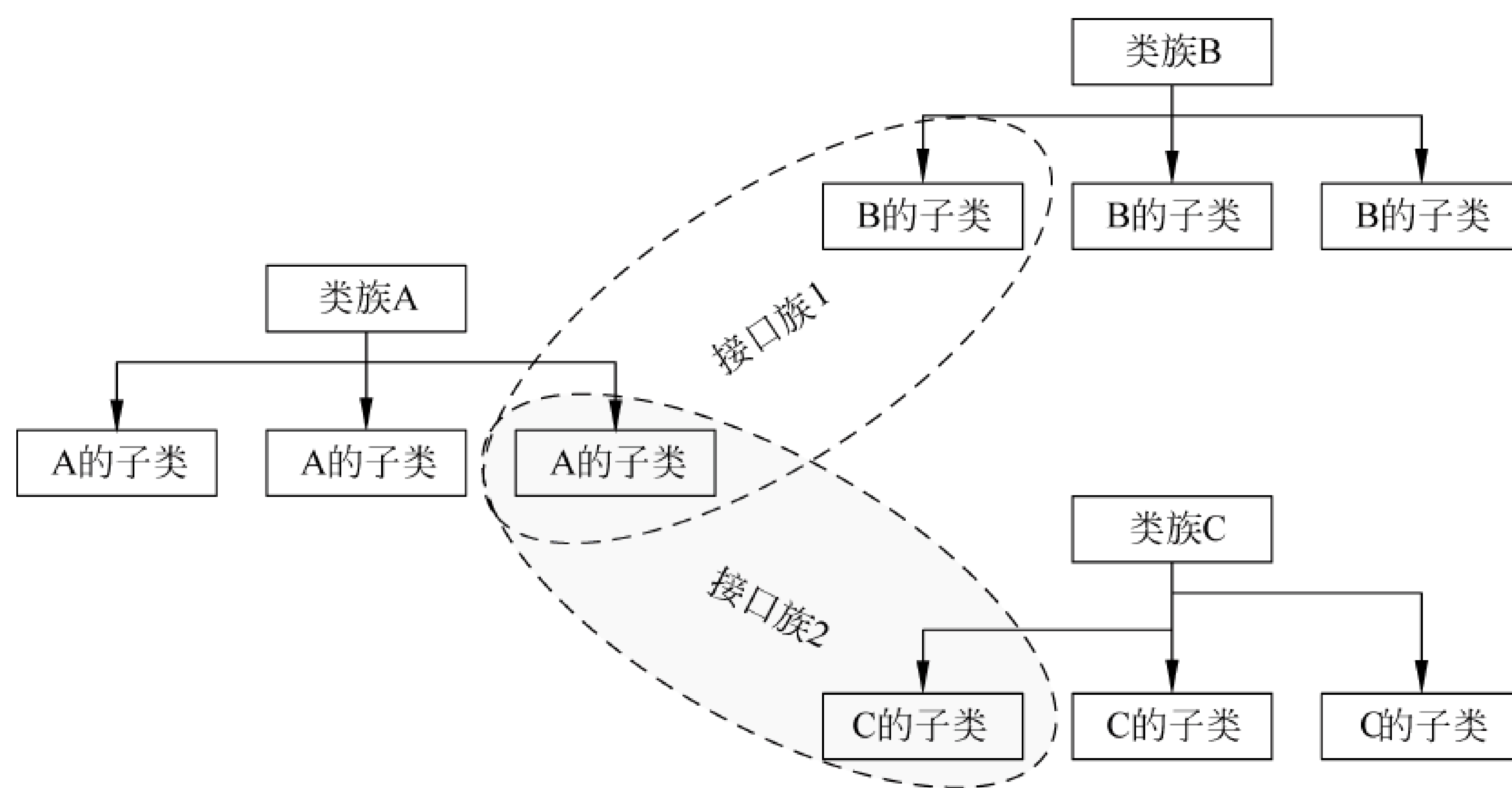


图 4-11 类族与接口族的关系示意图

(2) 接口族。

接口为类的组织管理又提供了一种新的形式。接口族中,各个类都实现了同样的接口,但它们之间的关系是松散的,而且没有类属关系。

利用对象替换与多态机制,同一接口族中的各个类之间也可以共用算法代码。例如,图 4-11 所示接口族 1 中,A 的子类和 B 的子类之间可以共用算法代码;接口族 2 中,A 的子类和 C 的子类之间也可以共用算法代码。

(3) 类族与接口族。

一个类只能在一个类族中,因为类只能单继承。但一个类可以在多个接口族中,因为它能同时实现多个接口。

本节习题

1. 下列关于继承与扩展的描述中,错误的是(　　)。
   A. 继承与扩展可以重用已有类的字段(属于数据代码)
   B. 继承与扩展可以重用已有类的方法(属于算法代码)
   C. 继承与扩展就是直接使用已有的类来定义对象
   D. 继承与扩展可以将多个类的共性部分提炼成超类,这样能减少重复代码
2. 下列关于抽象方法的描述中,正确的是(　　)。
   A. 抽象方法没有方法名　　B. 抽象方法没有返回值类型
   C. 抽象方法没有方法体　　D. 抽象方法没有形参列表
3. 下列关于抽象类的描述中,错误的是(　　)。
   A. 含有抽象方法的类称为抽象类,定义时必须使用关键字 abstract
   B. 不能使用抽象类创建对象,即抽象类不能实例化
   C. 不能定义抽象类的引用变量
   D. 抽象类可以作为超类定义子类

4. 接口是一种特殊的抽象类，其成员中不能包含(　　)。
   A. public 抽象方法　　B. public 静态方法
   C. protected 抽象字段　　D. public 静态只读字段
5. 类实现接口，其主要目的是(　　)。
   A. 继承接口中的方法成员　　B. 继承接口中的字段成员
   C. 继承接口中的方法签名　　D. 继承接口中的静态成员
6. 下列关于继承类和实现接口的描述中，正确的是(　　)。
   A. 类可以多继承，接口可以多实现　　B. 类可以多继承，接口只能单实现
   C. 类只能单继承，接口可以多实现　　D. 类只能单继承，接口只能单实现
7. 下列关于接口的描述中，错误的是(　　)。
   A. 接口是一种引用数据类型　　B. 接口可以用来定义引用变量
   C. 接口可以用来创建对象　　D. 接口引用变量可以引用其子类对象
8. 下列关于类族与接口族的描述中，错误的是(　　)。
   A. 类族中的各个类之间有类属关系　　B. 接口族中的各个类之间有类属关系
   C. 同一类族中的类可共用算法代码　　D. 同一接口族中的类可共用算法代码

4.6　4 种特殊的类定义形式

本节介绍 Java 语言中 4 种特殊的类定义形式，它们分别是内部类、局部类、匿名类和匿名方法。

4.6.1　内部类

在 Java 语言中，可以将一个类定义在另一个类的内部。定义在其他类内部的类称为**内部类**(inner class)。反过来，包含其他类的类称为**外部类**(outer class)。

内部类是一种特殊的类，它具有如下特点。

(1) 内部类是外部类的一个成员，可以访问所在外部类的私有成员。

(2) 内部类可以使用类成员所特有的权限(private、protected)进行管理。

(3) 在外部类之外的地方使用内部类，需通过外部类对象才能使用。

例 4-15 给出一个包含内部类 B 的外部类 AwithInner 示例代码。

例 4-15　一个包含内部类 B 的外部类 AwithInner 示例代码(AwithInner. java)

```
1  public class AwithInner {                    //外部类
2      public  int x = 10;                      //公有成员 x
3      private int y = 20;                      //私有成员 y
4
5      public class B {                         //内部类
6          public int z = 30;
7          public void bShow() {
8              System.out.println( x);          //内部类可以直接访问外部类的成员 x
9              System.out.println( y);          //内部类可以访问外部类的私有成员 y
10             System.out.println( z);
```

```
11          }  }
12
13          public void aShow() {               //在外部类中使用内部类 B
14              B bObj = new B();               //与使用普通的类一样
15              bObj.bShow();
16      }  }
```

内部类是作为外部类的一个成员进行管理的。如果内部类不是私有权限(private),则可以在外部类之外的其他地方使用这个内部类。下面给出一个在外部类 AwithInner 之外的其他地方使用内部类 B 的示例代码。

```
AwithInner aObj = new AwithInner();     //要想使用内部类,必须先定义外部类的对象
AwithInner.B bObj = aObj.new B();       //然后通过外部类的对象来 new 内部类的对象
bObj.bShow();
```

内部类可以应用于如下场合。

(1) 当只被某一个类使用的时候,可以将类定义成该类的内部类,并将内部类的访问权限设为私有权限(private)。

(2) 当希望访问某个类的私有成员时,可以将类定义成该类的内部类。

(3) 当希望将若干个具有关联关系的类分成一组进行管理时,可以将这些类集中定义到某个外部类中。

4.6.2 局部类

当只被某一个类使用的时候,可以将类定义成该类的内部类。当只被某个类中的某个方法使用的时候,可以将类定义成该方法中的**局部类**(local class)。例 4-16 给出一个局部类的示例代码。

例 4-16 一个包含局部类 B 的外部类 AwithLocal 示例代码(AwithLocal.java)

```
 1  public class AwithLocal {                   //外部类,其方法 aMethod 中包含一个局部类 B
 2      private int a = 10;                     //外部类的私有成员 a
 3      public void aMethod( int x) {           //方法中的形参 x
 4          final int y = 30;                   //方法中的局部只读变量(或称常量)y
 5
 6          class B {                           //定义在方法 aMethod 中的局部类
 7              int b = 40;                     //局部类的成员 b
 8              void showA()                    //访问外部类的成员 a
 9              {  System.out.println( a );  }
10              void showXY()                   //访问方法中的形参 x 和局部只读变量 y
11              {  System.out.println( x + " and " + y );  }
12              void showB()                     //访问本类的成员 b
13              {  System.out.println( b );  }
14          }
15
16          B obj = new B();                     //创建局部类 B 的对象 obj
17          obj.showA();  obj.showXY();  obj.showB();  //调用对象 obj 的方法
18  }  }
```

例 4-16 中的局部类 B 被定义在方法 aMethod()的内部，只能在这个方法中使用。局部类是一种特殊的类，它具有如下特点。

(1) 局部类访问外部类的成员时不受权限控制。

(2) 局部类可以访问所在方法的形参和局部变量，但要求被访问的形参和局部变量是常量(final)或事实常量(effectively final，数值在方法执行过程中不会改变)。

(3) 方法中的局部类和局部变量一样，不能(也不需要)设定访问权限。

(4) 局部类只能在某个方法内部使用，其他地方不需要使用这个类。

内部类、局部类可以继承超类或实现接口。例 4-17 给出一个实现接口的局部类 Java 示例代码。

例 4-17　一个实现接口的局部类 B 示例代码(在 4-16 的 AwithLocal.java 基础上修改而来)

```
 1  public class AwithLocal {                 //外部类，其方法 aMethod 中包含一个局部类 B
 2      private int a = 10;                   //外部类的私有成员 a
 3      public void aMethod( int x) {         //方法中的形参 x
 4          final int y = 30;                 //方法中的局部只读变量(或称常量)y
 5
 6          class B implements IShow {        //实现接口 IShow 的局部类 B
 7              int b = 40;                   //局部类的成员 b
 8              public void show()            //实现接口 IShow 里的抽象方法 show()
 9              {  System.out.println( a + " and " + b );  }
10          }
11
12          B obj = new B();                  //创建局部类 B 的对象 obj
13          obj.show();                       //调用对象 obj 的方法 show()
14  }  }
15
16  public interface IShow {                  //接口 IShow
17      public void show();                   //定义了一个抽象方法 show()
18  }
```

4.6.3　匿名类

当只被某个类中的某个方法使用的时候，可以将类定义成该方法中的局部类。如果这个局部类继承某个超类或实现某个接口，并且在方法中仅仅被使用一次，则可以省略局部类的类名，这就是**匿名类**(anonymous class)。匿名类必须继承某个超类或实现某个接口，在定义匿名类引用变量或创建匿名类对象时需使用这个超类或父接口的名字。例 4-18 给出一个实现接口的匿名类示例代码。

例 4-18　一个实现接口的匿名类示例代码(在 4-17 的 AwithLocal.java 基础上修改而来)

```
 1  public class AwithLocal {                 //外部类，其方法 aMethod 中包含一个匿名类
 2      private int a = 10;                   //外部类的私有成员 a
 3      public void aMethod( int x) {         //方法中的形参 x
 4          final int y = 30;                 //方法中的局部只读变量(或称常量)y
```

```
5
6        IShow obj = new IShow() {        //用一条语句完成定义匿名类和创建对象的工作
7            int b = 40;                  //匿名类的成员 b
8            public void show()           //实现接口 IShow 里的抽象方法 show()
9            { System.out.println( a + " and " + b ); }
10       };
11       obj.show();                      //调用对象 obj 的方法 show()
12 } }
13
14 public interface IShow {               //接口 IShow
15     public void show();                //定义了一个抽象方法 show()
16 }
```

匿名类是对仅用一次的内部类的简写形式。如果使用匿名类所定义的对象也只被使用一次，则还可以被进一步简写。例如，例 4-18 中的匿名类对象 obj 只被使用一次，代码第 6～11 行可简写为如下的形式：

```
(new IShow() {        //同时省略类名和对象名
        int b = 40;
        public void show()
        { System.out.println( b ); }
} ) . show() ;        //用一条语句完成下列工作：定义匿名类、创建其对象、调用该对象的方法
```

因为匿名类只用一次，因此可以省略类名。匿名类是一种特殊的类，它具有如下特点。

(1) 匿名类必须继承某个超类或接口，创建匿名类对象时使用其超类名或父接口名。其语法形式为：

```
new 超类名或父接口名() { 匿名类定义代码 } ;
```

定义匿名类引用变量时也需要使用其超类名或父接口名。例如：

```
IShow obj = new IShow() { … } ;
```

(2) 匿名类只能继承一个超类，或只能实现一个接口。

(3) 匿名类没有类名，无法定义构造方法。如果匿名类继承某个超类，则创建对象时将自动调用超类的构造方法(可以给定初始值初始化其中的超类成员)；如果匿名类只是实现某个接口，则不能进行初始化(父接口名后面的小括号里不能带参数)，因为接口本身也没有构造方法。

4.6.4 匿名方法

如果一个接口只包含一个抽象方法，则这样的接口称为**功能接口**(functional interface)。如果一个类只实现了某个功能接口，并且没有再定义任何其他成员，则该类将只包含一个方法成员，这样的类称为**功能类**(functional class)。

使用功能类所创建的对象，其中只包含一个方法成员。一个功能类对象，其本质就是一个完成某种程序功能的方法(即函数)。如果一个功能类对象只被使用一次，则可以使用 4.6.3 节介绍的匿名类来简化代码。例 4-19 给出一个功能接口和功能类的示例代码。

例 4-19 一个功能接口 IOperator 和功能类 AClass 的示例代码

```
   IOperator: 功能接口                          AClass: 实现接口 IOperator 的功能类
1  public interface IOperator {//功能接口       class AClass  implements IOperator {//功能类
2       int operation(int x, int y);//抽象方法      int operation(int x, int y)//实现接口里的方法
3  }                                                {  return x + y;  }    //加法运算
4                                               }
```

创建例 4-19 中功能类 AClass 的对象，可以使用如下 3 种方法。

(1) 正常方法。

```
AClass r = new AClass();                    //使用类 AClass 创建一个功能类对象
System.out.println( r.operation(3, 8) );    //调用对象的方法 operation(),显示结果: 11
```

(2) 改用匿名类简化代码(用匿名类取代例 4-19 中 AClass)。

```
IOperator r = new IOperator() {             //用一条语句完成定义类和创建对象的工作
    int operation(int x, int y)             //实现接口里的方法
    {  return x + y;  }                     //加法运算
};
System.out.println( r.operation(3, 8) );  //调用对象的方法,显示结果: 11
```

(3) 改用匿名方法进一步简化代码(用匿名方法取代 2)中的匿名类)。

```
IOperator r = (int x, int y ) ->     //省略了匿名类中的运算符 new、接口名、方法名及其类型
{  return x + y;   }                 //只保留方法的形参列表和方法体
System.out.println( r.operation(3, 8) );  //调用对象的方法,显示结果: 11
```

上述匿名方法还可以进一步简写为:

```
IOperator r = (x, y) ->{  return x + y;  };  //进一步省略形参的数据类型
```

或

```
IOperator r = (x, y) ->x + y;          //只有一条语句时可进一步省略大括号和 return
```

Java 语言将省略掉方法名和返回值类型的方法称为**匿名方法**(anonymous method)。匿名方法只保留了方法的形参列表和方法体，并在形参列表和方法体之间插入一个**指向运算符**“->”。匿名方法看起来像是一个表达式，术语称为 **Lambda 表达式**。

实际 Java 编程中，程序员在创建一次性使用的对象时，常常喜欢以匿名类或匿名方法的形式来简化程序代码。

本节习题

1. 下列不同场合中，(　　)不应该使用内部类。
   A. 当只被某一个类使用的时候，可以将类定义成该类的内部类
   B. 当希望访问某个类的私有成员时，可以将类定义成该类的内部类
   C. 当希望将若干个类归成一组管理时，可以将它们集中定义到某个外部类中
   D. 当一个类被广泛使用的时候，可以将该类定义成某个类的内部类

2. 下列关于局部类的描述中，错误的是(　　)。
   A. 局部类访问外部类的成员时不受权限控制
   B. 局部类可以访问所在方法中的所有形参和局部变量
   C. 方法中的局部类和局部变量一样，不能设定访问权限
   D. 局部类只能在某个方法内部使用，其他地方不需要使用这个类
3. 下列关于匿名类的描述中，错误的是(　　)。
   A. 省略掉类名的局部类被称为匿名类
   B. 匿名类只能继承一个超类
   C. 匿名类必须继承某个超类或实现某个接口
   D. 匿名类可以实现多个接口
4. 下列关于匿名方法的描述中，错误的是(　　)。
   A. 省略掉方法名和返回值类型的方法称为匿名方法
   B. 匿名方法具有形参列表
   C. 匿名方法具有方法体
   D. 匿名方法是一个抽象方法，即只有方法签名
5. 使用匿名类或匿名方法的目的是(　　)。
   A. 提高程序代码的运行速度
   B. 提高程序代码的可读性
   C. 提高程序代码的可重用性
   D. 简化程序代码

本章学习要点

- 学会使用组合和继承的方法来定义新类，这样可以提高类代码的开发效率。
- 应从提高算法代码重用性的角度去理解对象的替换与多态机制。
- 熟练掌握接口的定义和实现方法，并充分理解接口与超类的区别。
- 熟练掌握匿名类和匿名方法的简写形式。

本章习题

1. 编写程序。使用组合的方法编写一个计算圆环面积的 Java 程序。要求：先设计一个圆形类 Circle，再基于类 Circle 使用组合的方法定义一个圆环类 Ring。圆环可认为是由一大一小两个同心圆组合而成。最后编写一个测试类测试所编写的圆环类 Ring。

2. 编写程序。使用继承与扩展的方法编写一个计算圆环面积的 Java 程序。要求：先设计一个圆形类 Circle，再通过继承类 Circle 定义一个圆环类 Ring。圆环可认为是在圆形基础上再增加一个描述线宽的属性，即带边框的圆形。最后编写一个测试类测试所编写的圆环类 Ring。

3. 编写程序。完成以下的对象替换与多态实验。

(1) 编写一个描述鸡的类 Chicken(属性至少包括鸡的名字和质量，方法至少包括构造方法、显示鸡的质量、模拟鸡的叫声等)。

(2) 继承类 Chicken，分别定义描述小鸡的类 Chick、描述母鸡的类 Hen、描述公鸡的类 Cock，重写模拟鸡叫的方法。

要求：运行下面的测试类示例代码，应该得到给定的运行结果。

```
public class JChickenTest {                          //测试类
    public static void main(String[ ] args) {        //主方法
        Chicken p[ ] = new Chicken[4];
        //定义 4 只鸡：鸡(名 C1)、小鸡(名 C2)、母鸡(名 C3)、公鸡(名 C4)
        char c1[ ] = { 'C', '1' };  p[0] = new Chicken(c1, 1);
        char c2[ ] = { 'C', '2' };  p[1] = new Chick(c2, 0.2);
        char c3[ ] = { 'C', '3' };  p[2] = new Hen(c3, 1.5);
        char c4[ ] = { 'C', '4' };  p[3] = new Cock(c4, 2.5);
        for (int n = 0; n < 4; n++)  show_sing( p[n] );
        return;
    }
    public static void show_sing(Chicken c) {
        c.show();  c.sing();
    }
}
```

运行测试类示例代码应该能得到如下运行结果。

```
C1:1.0kg
C1:鸡叫了
C2:0.2kg
C2:小鸡,叽叽叽
C3:1.5kg
C3:母鸡,咯咯嗒
C4:2.5kg
C4:公鸡,咕咕咕
```

4. 重写程序。阅读并理解 4.6 节例 4-15～例 4-19 中内部类、局部类、匿名类和匿名方法的 Java 示例程序，然后重写这些程序。

第3部分　Java应用程序开发

- 第5章　Java基础类库
- 第6章　图形用户界面程序
- 第7章　输入输出流
- 第8章　多线程并发编程
- 第9章　网络编程
- 第10章　数据库编程

第5章 Java基础类库

Java 语言经过二十多年的发展，已经积累了大量编写好的、可实现各种不同功能的类。Java 语言将这些类打包起来，以类库的形式提供给广大程序员使用。这些由 Java 语言自己所提供的类库统称为 **Java API**(Application Programming Interface)，参见图 5-1。

<table>
<tr><td>Swing</td><td>Java 2D</td><td>AWT</td><td colspan="2">Accessibility</td><td rowspan="6">扩展类库</td></tr>
<tr><td>Drag and Drop</td><td>Input Methods</td><td>Image I/O</td><td>Print Service</td><td>Sound</td></tr>
<tr><td>IDL</td><td>JDBC</td><td>JNDI</td><td>RMI</td><td>RMI-IIOP</td><td>Scripting</td></tr>
<tr><td>Beans</td><td>Security</td><td>Serialization</td><td colspan="2">Extension Mechanism</td></tr>
<tr><td>JMX</td><td>XML JAXP</td><td>Networking</td><td colspan="2">Override Mechanism</td></tr>
<tr><td>JNI</td><td>Date and Time</td><td>Input/Output</td><td colspan="2">Internationalization</td></tr>
<tr><td colspan="5">lang and util</td><td rowspan="4">基础类库</td></tr>
<tr><td>Math</td><td>Collections</td><td>Ref Objects</td><td colspan="2">Regular Expressions</td></tr>
<tr><td>Logging</td><td>Management</td><td>Instrumentation</td><td colspan="2">Concurrency Utilities</td></tr>
<tr><td>Reflection</td><td>Versioning</td><td>Preferences API</td><td>JAR</td><td>ZIP</td></tr>
<tr><td colspan="5">Java Hotspot Client and Server VM</td></tr>
</table>

图 5-1　Java API 整体架构(摘自：Java Platform Standard Edition 8 Documentation)

类库相当于已经编写好的**程序零件**。重用类库中的类，相当于用现成的零件来组装程序，这样就能使程序员快速开发出各种功能强大的软件。用已有的程序零件来组装自己的软件产品，这就是程序的**应用开发**。

要想使用 Java API 类库中的类，首先要了解 Java API 中有哪些常用的包，每个包里又有哪些类，这些类的功能是什么，然后再深入学习每个类，了解类里有哪些成员，各成员的功能和用法是什么。通过不断学习和探索，逐步梳理清楚 Java API 类库的整体脉络。

了解 Java API 的最主要途径是学习 Java 官网上发布的 Java API 说明文档。它是 Java API 最权威的学习资料，为程序员提供了详细的帮助信息。学会阅读 Java API 说明文档，这也是 Java 语言程序设计学习的一部分。

本书前 4 章学习了 Java 基础语法和面向对象程序设计方法。后续章节主要学习如何利用 Java API 来进行程序的应用开发，内容包括：

- 学习阅读 Java API 说明文档的基本方法。
- 了解常见的编程应用场景，掌握 Java API 中相关类的使用方法。

• 探索 Java API 类库，循序渐进，从整体上把握 Java API 类库的架构。

在后续章节中，读者将接触到大量具体的程序应用场景和案例。后续章节的学习过程既是 Java 知识积累的过程，同时也是自学能力培养的过程。日积月累，化蛹成蝶，相信读者最终都能够独立开启自己的 Java 探索之旅。

5.1 数学类 Math

Java API 是一个类的海洋。探索 Java API 类库，从数学类 Math 开始(见图 5-2)。

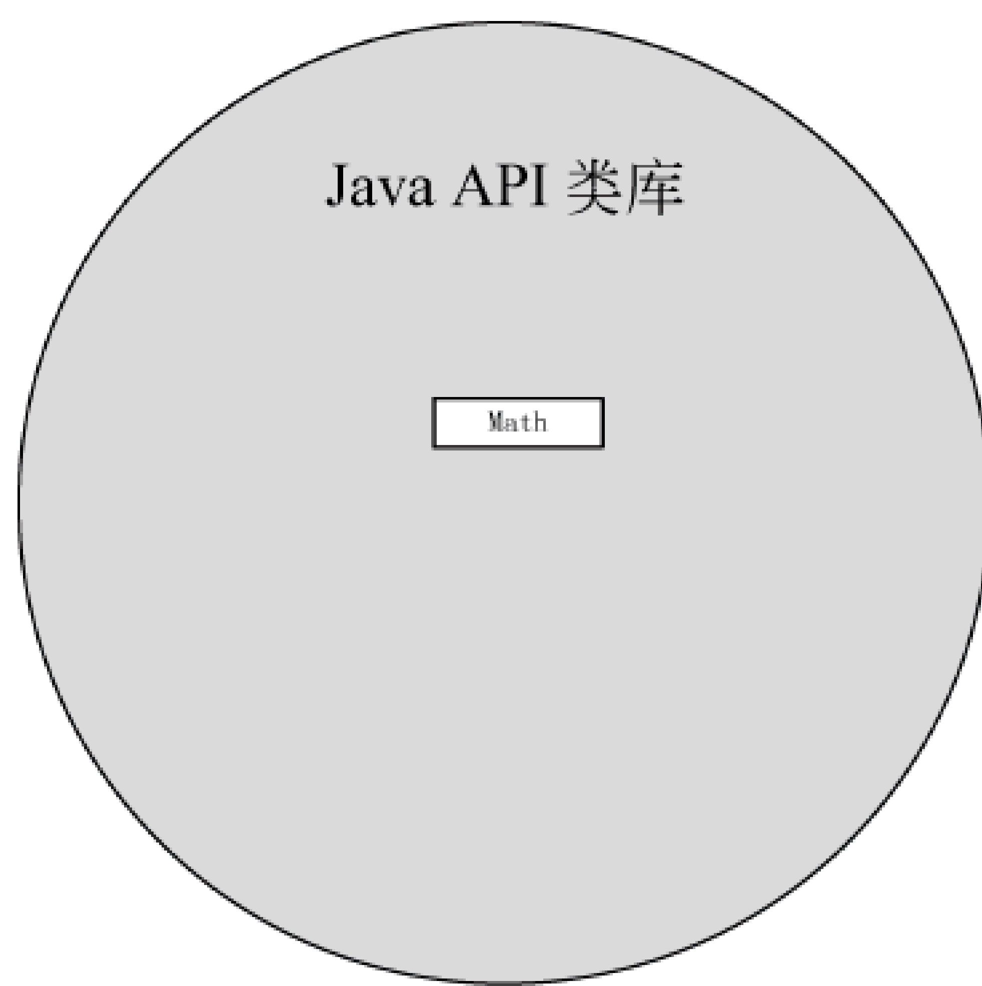


图 5-2 探索 Java API 类库从数学类 Math 开始

了解数学类 Math，从 Java API 说明文档开始。Java API 说明文档对数学类 Math 有如下描述(注：Java API 说明文档是英文的，没有中文翻译)。

The class **Math** contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

这段英文的中文翻译是：类 Math 包含一组数值运算方法，例如指数、对数、平方根和三角函数等初等函数。

Java API 说明文档对每个类都有详细的说明，其中包括类的简要介绍、类包含哪些成员，以及各成员的功能和用法。本书将其中的关键信息摘录下来，以方便读者阅读。首先请读者简单浏览一下数学类 Math 的说明文档。

java. lang. **Math** 类说明文档(摘自：Java Platform Standard Edition 8 Documentation，以下同)

public final class **Math**
extends **Object**

	修 饰 符	类成员(节选)	功 能 说 明
1	static	double **E**	字段，数学常数 e
2	static	double **PI**	字段，数学常数 π
3	static	int **abs**(int a)	求 int 型整数的绝对值
4	static	double **abs**(double a)	求 double 型实数的绝对值
5	static	double **sin**(double a)	求正弦值
6	static	double **cos**(double a)	求余弦值
7	static	double **log**(double a)	求自然对数
8	static	double **log**10(double a)	求以 10 为底的对数
9	static	double **pow**(double a, double b)	求 a 的 b 次方
10	static	double **random**()	返回一个 0～1 的随机数
11	static	double **sqrt**(double a)	求平方根
12	static	double **toDegrees**(double angrad)	将弧度转换为角度
13	static	double **toRadians**(double angdeg)	将角度转换为弧度

…

5.1.1 阅读 Java API 类的说明文档

在大致了解了类的功能之后,读者需要仔细阅读类的说明文档。类有 3 个学习要点,分别是类的**全称**、**定义**及其中的各个**成员**。

1. 类的全称

从数学类 Math 的全称 java.lang.Math 中可以读出哪些信息?

- **java**:包名。
- **lang**:java 包下的子包名。
- **Math**:java 包下 lang 子包里的类名。

Java API 有一套自己的命名风格,**包名**、**子包名**和**方法名**都以小写英文字母开头,而**类名**则是以大写英文字母开头的。建议读者在编写自己的 Java 程序时参考使用这种命名风格,这样可以在今后阅读程序时能更容易地区分出包名、类名和方法名。

java.lang 子包是 Java API 中最基础的**语言包**,每个 Java 程序都需要用到这个包。为了方便程序员,Java 编译器会为每个 Java 程序都自动导入这个语言包,这相当于是为程序添加了如下 import 语句:

```
import java.lang.* ;                    //导入 Java 语言包里的所有类
```

因此,程序员在使用 Java 语言包里的数学类 Math 时可以直接用类名 Math,不需要写全称 java.lang.Math。

2. 类的定义

Java API 说明文档给出的数学类 Math 定义如下:

```
public final class Math
extends Object
```

从这个数学类 Math 的定义中可以读出哪些信息?

- **public**:Math 是一个公有类,任何 Java 程序都可以使用这个类。
- **final**:Math 是一个最终类,不能被继承、扩展。
- **extends**:Math 继承了一个超类,这个超类的名字叫 **Object**。

3. 类的成员

数学类 Math 包含很多成员,其中有字段 E、PI 等数学常数,还有方法 abs()、sin()等数学函数。阅读说明文档可以清楚地了解各类成员的功能和用法。

1) 类成员的功能和用法

举例:int abs(int a)

这是数学类 Math 中的一个方法成员。阅读方法成员说明文档时要关注其功能是什么,方法名、形参、返回值类型分别是什么。至于这个方法是怎么编的,方法体是什么,读者没必要关心。这个方法的功能是什么,怎么用?这才是读者应该关心的。

```
double x = Math.abs( -8 );                    //方法 abs(int a)的功能是求 int 型整数的绝对值
```

2) static 成员

类成员前面的修饰符 static 表示静态成员。数学类 Math 中的成员全部都是静态成员。访问静态成员不需要定义对象,可以直接通过类名访问。例如 Math.**PI**、Math.**abs**(−8)。

3) 类成员的访问权限

类成员的访问权限有 4 种,分别是公有权限(public)、保护权限(protected)、私有权限(private)和默认权限(未指定访问权限)。

在 Java API 中,只有类的公有成员和保护成员是提供给 Java 程序员使用的,而私有成员和默认权限成员则是 Java API 内部使用的。因此,Java API 说明文档不会对私有成员或默认权限成员进行说明,因为这两种成员本来就不是给 Java 程序员使用的,它们是隐藏的。

5.1.2 编写测试程序来学习 Java API 类

在阅读了 Java API 类的说明文档之后,程序员需要编写测试程序来练习类的具体用法。例 5-1 给出一个数学类 Math 的测试程序示例代码。

例 5-1 一个数学类 Math 的测试程序示例代码(MathTest.java)

```
 1  public class MathTest {                                          //测试类
 2      public static void main(String[] args) {                     //主方法
 3          System.out.println( Math.abs( -8 ) );                    //求绝对值
 4          System.out.println( Math.sqrt( 16 ) );                   //求平方根
 5          System.out.println( Math.sin( Math.PI/2 ) );             //求正弦值
 6          System.out.println( Math.toDegrees( Math.PI ) );         //将弧度换算成角度
 7          System.out.println( Math.random() );                     //取一个随机数
 8          System.out.println( Math.random() );                     //再取一个随机数
 9      }
10  }
```

在 Eclipse 集成开发环境中输入并运行例 5-1 的程序,运行结果如图 5-3 所示。将**运行结果与程序源代码**进行比对,可以帮助程序员准确理解类中各成员的功能和用法。**强调一下**:编写测试程序,这是学习 Java API 最有效的方法。



图 5-3 例 5-1 程序的运行结果

本节习题

1. 数学类 Math 的全称 java. lang. Math 中不包含(　　)信息。
   A. 包名　　B. 子包名
   C. 类名　　D. 超类名
2. Java API 说明文档给出的数学类 Math 定义如下：

```
public final class Math
extends Object
```

这个类定义中不包含(　　)信息。
   A. 包名　　B. 类名
   C. 超类名　　D. 类的访问权限
3. Java API 说明文档给出数学类 Math 的方法成员 sin()定义如下：

```
static double sin(double a)
```

这个方法成员定义中不包含(　　)信息。
   A. 方法名　　B. 形参列表
   C. 方法体　　D. 返回值类型
4. 如果 Java API 说明文档没有给出类成员的访问权限，则该类成员的权限是(　　)。
   A. public　　B. private
   C. protected　　D. 默认权限
5. 数学类 Math 中返回随机数的方法是(　　)。
   A. sin()　　B. sqrt()
   C. random()　　D. Random()

5.2 字符串类

Java API 将字符数组和字符串处理方法封装成字符串类。字符串类的功能更强，使用更方便。常用的字符串类有如下 3 个。

(1) 字符串类 **String**。用于存储和处理字符串常量。

(2) 可变字符串类 **StringBuilder**。适用于存储和处理字符串变量，主要用于文字处理和编辑。

(3) 多线程可变字符串类 **StringBuffer**。与 StringBuilder 功能相同，但可用于多线程程序。其字符串处理算法要复杂一些，运行速度相对也要慢一些。

注：多线程程序将在第 8 章讲解。

5.2.1 字符串类 String

请读者阅读下面的字符串类 String 说明文档。

java. lang. **String** 类说明文档

public final class **String**
extends **Object**
implements **Serializable**，**Comparable**<String>，**CharSequence**

	修饰符	类成员(节选)	功 能 说 明
1		**String**()	无参构造方法
2		**String**(char[] value)	有参构造方法
3		**String**(String original)	拷贝构造方法
4		int **length**()	返回字符串长度
5		char **charAt**(int index)	返回指定下标的字符
6		int **indexOf**(int ch)	返回指定字符 ch 在字符串中第一次出现的下标
7		int **indexOf**(String str)	返回子字符串 str 在字符串中第一次出现的下标
8		int **compareTo**(String anotherString)	与另一个字符串比较大小(按字母顺序)
9		int **compareToIgnoreCase**(String str)	与另一个字符串比较大小(不区分大小写)
10		String **substring**(int beginIndex, int endIndex)	取出指定下标区间里的子字符串
11		String **replace**(CharSequence target, CharSequence replacement)	替换子字符串
12		String[] **split**(String regex)	分割字符串
13		boolean **matches**(String regex)	检查是否与正则表达式匹配
14		String **concat**(String str)	将字符串 str 连接到当前字符串末尾,生成一个新字符串
15		String **toLowerCase**()	将字符串中的大写英文字母转换成小写
16		String **toUpperCase**()	将字符串中的小写英文字母转换成大写
17		String **trim**()	去除字符串两端的空格
18		byte[] **getBytes**()	按字符编码转换成字节数组
19		byte[] **getBytes**(Charset charset)	按指定编码转换成字节数组
20	static	String **format**(String format, Object…args)	生成格式化字符串
21	static	String **valueOf**(int i)	将整数转换成字符串
22	static	String **valueOf**(long l)	将长整数转换成字符串
23	static	String **valueOf**(float f)	将单精度浮点数转换成字符串
24	static	String **valueOf**(double d)	将双精度浮点数转换成字符串
25	static	String **valueOf**(char[] data)	将字符数组转换成字符串

…

1. 类的全称

从字符串类 String 的全称 java. lang. String 中可以读出哪些信息?

- **java**：包名。
- **lang**：java 包下的子包名。

- **String**：java 包下 lang 子包里的类名。

对比字符串类 String 和 5.1 节的数学类 Math，可以发现这两个类是在同一个包里的，即 java.lang 语言包。这个包存放了 Java API 中最基础的类和接口。

2. 类的定义

Java API 说明文档给出的字符串类 String 定义如下：

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

从这个类定义中可以读出哪些信息?

- **public**：String 是一个公有类，任何 Java 程序都可以使用这个类。
- **final**：String 是一个最终类，不能被继承、扩展。
- **extends**：String 继承了一个超类，这个超类的名字叫 **Object**。
- **implements**：String 类还实现了 3 个接口，分别是 **Serializable**（可序列化的）、**Comparable**（可比较的）、**CharSequence**（字符序列）。

对比字符串类 String 和 5.1 节的数学类 Math，可以发现这两个类继承的是同一个超类，即 Object。但字符串类 String 比数学类 Math 还多实现了 3 个接口。

3. 字符串类 String 的用法

下面简单介绍一些字符串类 String 的基本用法。

1）定义字符串对象

字符串类 String 包含多个构造方法，因此使用该类定义对象时可以有多种不同的初始化形式。例如：

```
String s1 = new String();               //未初始化
String s2 = new String( "China 中国" );  //用字符串常量进行初始化
String s3 = new String( s2 );           //用已有的字符串对象进行初始化
```

2）操作字符串对象

字符串类 String 有很多字符串处理方法。例如：

```
System.out.println( s2.length() );      //求 s2 的字符串长度：s2 的长度为 7
System.out.println( s2.substring(1, 3) ); //取 s2 的子串：返回 1～3 的子串"hi"(不含 3)
```

3）字符串可以相加

可以使用加号“+”连接两个字符串。例如：

```
String s = s2 + ",你好";               //连接两个字符串，并赋值给 s
s += ",世界";                          //在 s 后面再连接一个字符串",世界"
```

4）修改字符串

String 类的字符串对象在内容改变时总是会生成一个新的字符串对象。例如：

```
String s = new String ("abc" );        //引用变量 s 指向字符串对象"abc"
```

```
s += "ef";                    //内存中将会有两个独立的字符串对象"abc"和"abcef"
```

String 类的字符串对象相加，不是在原字符串后面添加内容，而是相加后生成一个的新字符串对象"abcef"。引用变量 s 会改变指向，引用这个新生成的字符串。原字符串"abc"保持不变。字符串类 String 用于存储字符串常量，因此也称为不可变字符串类。

5) 比较两个字符串的大小

可以用方法 compareTo()来比较两个字符串的大小。例如：

```
String s = new String("cd");
System.out.println( s.compareTo("ab") );   //方法 compareTo()返回一个正数,因为"cd"大于"ab"
System.out.println( s.compareTo("cd") );   //方法 compareTo()返回 0,因为"cd"和"cd"相等
System.out.println( s.compareTo("ef") );   //方法 compareTo()返回一个负数,因为"cd"小于"ef"
```

6) 静态方法 format()

字符串类 String 提供了一个生成格式化字符串的静态方法 format()，其用法非常像 C 语言里的 printf()或 sprintf()。例如：

```
int x = 5;  double y = 16.8;
String s = String.format("x = %d, y = %5.2f", x, y);  //生成格式化字符串
System.out.println( s );                 //显示字符串 s,显示结果: x = 5, y = 16.80
```

4. 字符串类 String 的测试程序

例 5-2 给出一个字符串类 String 的测试程序示例代码。

例 5-2　一个字符串类 String 的测试程序示例代码(StringTest.java)

```
 1  public class StringTest {                                  //测试类
 2      public static void main(String[] args) {              //主方法
 3          int x = 5;  double y = 16.8;
 4          String s = String.format("x = %d, y = %5.2f", x, y); //格式化字符串
 5          System.out.println( s );                           //显示字符串 s
 6          //下面演示字符串对象的定义与处理
 7          String s1 = new String();                          //先定义 3 个字符串对象
 8          String s2 = new String("Abcd");
 9          String s3 = new String("Abcd cde");
10          System.out.println( s1.length() );                 //空字符串的长度为 0
11          System.out.println( s2.length() );                 //s2 的长度为 4
12          System.out.println( s2.toUpperCase() );            //返回一个大写的字符串
13          System.out.println( s3.indexOf("cd") );            //返回 cd 第一次出现的位置
14          System.out.println( s3.substring(1, 3) );          //返回 1~3 的子字符串
15          System.out.println( s3.concat("fg") );             //连接: s3 + "fg"
16          System.out.println( s3 + "fg" );                   //连接: s3 + "fg"
17  } }
```

在 Eclipse 集成开发环境中运行例 5-2 的程序，运行结果如图 5-4 所示。

5. 接口 Comparable

Comparable 是 Java API 定义的一个**可比较大小的**接口。



图 5-4　例 5-2 程序的运行结果

```
public interface Comparable<T> {
    int compareTo(T o);                //比较两个对象大小的抽象方法
}
```

字符串类 String 实现了这个接口，具体实现了比较两个字符串大小的算法，因此可以用方法 compareTo()来比较字符串的大小。方法 compareTo()返回一个整数值，用负数、零、正数分别表示小于、等于和大于。**注：**< T >是泛型编程，将在 5.7 节讲解。

任何一个类都可以实现 Comparable 接口，然后编写 compareTo()方法，具体实现比较两个对象大小的算法。

5.2.2　可变字符串类 StringBuilder

字符串类 String 用于存储字符串常量。可变字符串类 StringBuilder 可存储字符串变量。StringBuilder 类中的字符串可以追加(append)、插入(insert)、替换(replace)和删除(delete)，这些修改属于"原地修改"。

StringBuilder 类有**长度**(length)和**容量**(capacity)两个概念。长度是指实际存储的字符个数，而容量则是指最多能存储的字符个数。当要保存的字符串长度大于容量时，StringBuilder 类将自动增加容量(即分配更多的内存)。StringBuilder 类相当于可变长的字符数组。请读者阅读下面的可变字符串类 StringBuilder 说明文档。

java. lang. **StringBuilder** 类说明文档

public final class **StringBuilder**
extends **Object**
implements **Serializable**，**CharSequence**

	修饰符	类成员(节选)	功 能 说 明
1		**StringBuilder**()	无参构造方法
2		**StringBuilder**(String str)	有参构造方法
3		**StringBuilder**(int capacity)	有参构造方法
4		StringBuilder **append**(CharSequence s)	追加一个字符串
5		StringBuilder **append**(char[] str)	追加一个字符数组
6		StringBuilder **append**(int i)	追加一个整数(转换成字符串)
7		StringBuilder **append**(double d)	追加一个实数(转换成字符串)
8		StringBuilder **insert**(int dstOffset，CharSequence s)	插入一个字符串

续表

	修饰符	类成员(节选)	功 能 说 明
9		StringBuilder **insert**(int offset, int i)	插入一个整数(转换成字符串)
10		StringBuilder **insert**(int offset, double d)	插入一个实数(转换成字符串)
11		StringBuilder **replace**(int start, int end,String str)	替换子字符串
12		StringBuilder **delete**(int start, int end)	删除子字符串
13		int **indexOf**(String str)	查找子字符串
14		String **substring**(int start, int end)	返回子字符串
15		int **length**()	返回字符串长度
16		int **capacity**()	返回字符容量
		…	

例 5-3 给出一个可变字符串类 StringBuilder 的测试程序示例代码。

例 5-3 一个可变字符串类 StringBuilder 的测试程序示例代码(SBuilderTest.java)

```
 1  public class SBuilderTest {                               //测试类
 2      public static void main(String[] args) {              //主方法
 3          String Builder s = new StringBuilder( 100 );//字符容量为 100 个字符(单线程)
 4          //String Buffer s = new StringBuffer( 100 ); //类 StringBuffer 可支持多线程
 5          s.append("helloChina");                           //追加字符
 6          System.out.println( s.toString() );               //显示所返回字符串内容
 7          System.out.println( s.length() );                 //显示字符串长度
 8          System.out.println( s.capacity() );               //显示字符容量
 9          s.setCharAt(0, 'H');                              //设定指定位置的字符
10          System.out.println( s );                          //可以省略".toString()"
11          s.replace(5, 10, "中国");                         //将 5~10 的字符替换成"中国"
12          System.out.println( s );
13          System.out.println( s.length() );                 //显示新字符串长度
14          s.insert(5, ',');  System.out.println( s );       //插入一个字符",",然后显示内容
15          s.delete(5, 8);  System.out.println( s );         //删除 5~8 的字符,然后显示内容
16  }  }
```

在 Eclipse 集成开发环境中运行例 5-3 的程序,运行结果如图 5-5 所示。

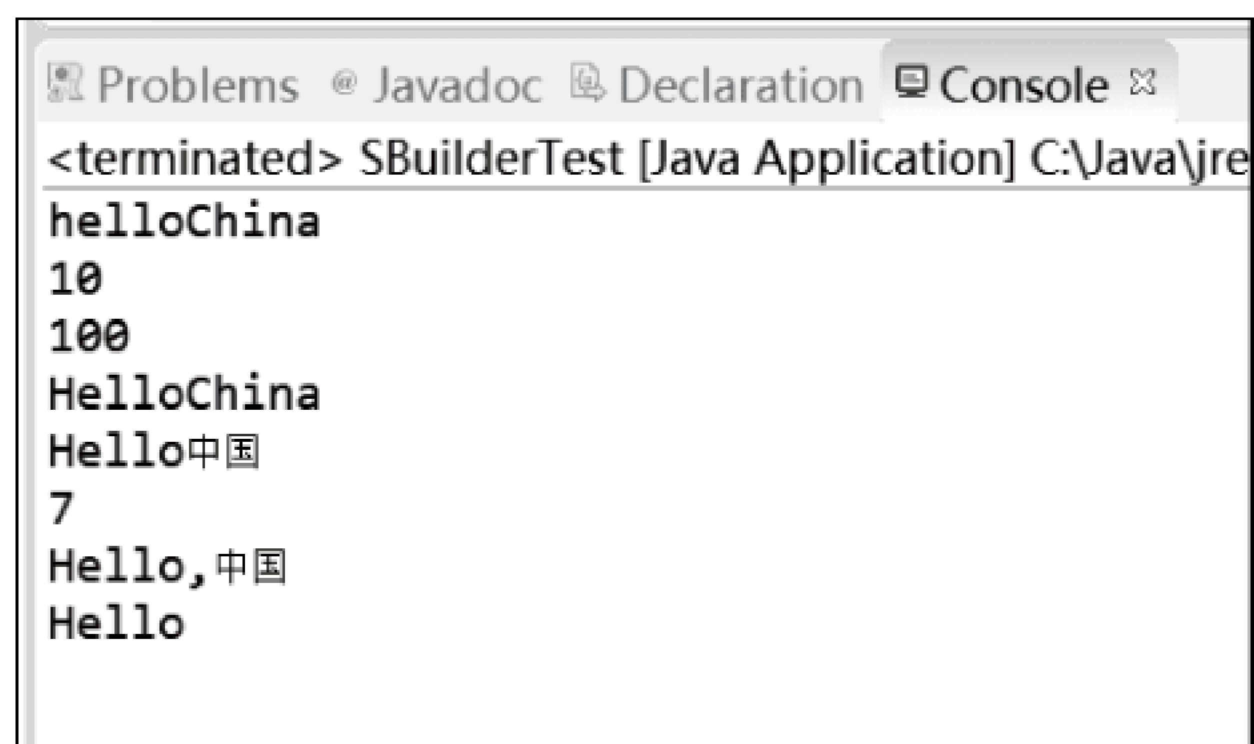


图 5-5 例 5-3 程序的运行结果

本节习题

1. 下面的类(　　)不是 Java API 中的字符串类。
   A. String　　B. StringBuilder　　C. StringBuffer　　D. Character
2. 字符串类 String 中取出指定位置字符的方法是(　　)。
   A. charAt()　　B. getBytes()　　C. substring()　　D. valueOf()
3. 字符串类 String 中取出某个位置区间内子字符串的方法是(　　)。
   A. charAt()　　B. getBytes()　　C. substring()　　D. valueOf()
4. 字符串类 String 中生成格式化字符串的静态方法是(　　)。
   A. print()　　B. println()　　C. compareTo()　　D. format()
5. 可变字符串类 StringBuilder 中替换某个位置区间内子字符串的方法是(　　)。
   A. append()　　B. insert()　　C. replace()　　D. delete()

5.3　基本数据类型的包装类

Java 语言预定义了 8 种基本数据类型。Java API 又将这 8 种基本数据类型以类的语法形式分别包装起来，定义了 8 个类。这 8 个类称为基本数据类型的**包装类**(wrapper class)，见表 5-1。

表 5-1　基本数据类型及其包装类

基本数据类型	byte	short	int	long	float	double	char	boolean
包装类	Byte	Short	Integer	Long	Float	Double	Character	Boolean

包装类具有与基本数据类型相关的常量和处理方法。常量主要包括基本数据类型的最大值和最小值、占用字节数等。处理方法主要包括比较大小、类型转换等。表 5-1 中 8 个包装类的用法基本相同，本节以整数类 Integer 为例进行讲解。

1. 整数类 Integer

请读者阅读下面的整数类 Integer 说明文档。

java. lang. **Integer** 类说明文档

public final class **Integer**
extends **Number**
implements **Comparable** < Integer >

	修饰符	类成员(节选)	功 能 说 明
1	static	int **BYTES**	字段，int 型占用字节数
2	static	int **MAX_VALUE**	字段，int 型的最大值
3	static	int **MIN_VALUE**	字段，int 型的最小值
4		**Integer**(int value)	构造方法

续表

	修饰符	类成员(节选)	功 能 说 明
5		**Integer**(String s)	构造方法
6		int **compareTo**(Integer anotherInteger)	与另一个 Integer 对象比较大小
7		int **intValue**()	将 Interger 转换成 int
8		String **toString**()	将 Interger 转换成 String
9	static	int **parseInt**(String s)	将字符串转换成 int
10	static	Integer **valueOf**(int i)	将 int 转换成 Integer
11	static	String **toString**(int i)	将 int 转换成字符串
		…	

例 5-4 给出一个整数类 Integer 的测试程序示例代码。

例 5-4 一个整数类 Integer 的测试程序示例代码(WrapperTest.java)

```
1  public class WrapperTest {                                                  //测试类
2      public static void main(String[] args) {                               //主方法
1          System.out.println(Integer.BYTES);                                  //int 型的字节数
4          System.out.println(Integer.MIN_VALUE);                              //int 型的最小值
5          System.out.println(Integer.MAX_VALUE);                              //int 型的最大值
6          //下面演示比较两个 Integer 对象的大小
7          Integer iObj1 = new Integer(20);                                    //初始化为 20
8          Integer iObj2 = new Integer("30");                                  //初始化为 30
9          System.out.println( iObj1 > iObj2 );                                //比较大小
10         System.out.println( iObj1.compareTo(iObj2) );                       //比较大小
11         //下面演示 Integer 与其他类型之间的转换
12         int i = iObj2.intValue();  System.out.println( i );                 //将 Interger 转换成 int
13         Integer iObj3 = Integer.valueOf( i + 10 );                          //将 int 转换成 Integer
14         System.out.println( iObj3.toString() );                             //将 Integer 转换成 String
15         i = Integer.parseInt("50");  System.out.println( i );               //将字符串转换成 int
16         System.out.println( Integer.toString(i) );                          //将 int 转换成字符串
17 } }
```

在 Eclipse 集成开发环境中运行例 5-4 的程序,运行结果如图 5-6 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> WrapperTest [Java Application] C:\Java
4
-2147483648
2147483647
false
-1
30
40
50
50
```

图 5-6 例 5-4 程序的运行结果

2. 数值类 Number

从整数类 Integer 的说明文档可以看出，整数类 Integer 继承了一个超类，即数值类 Number。实际上，表 5-1 的 8 个包装类中除了 Character 和 Boolean，剩下的 6 个都是数值类 Number 的子类。阅读数值类 Number 的说明文档发现，这个类也是从超类 Object 继承而来的。

java. lang. **Number** 类说明文档

public abstract class **Number**
extends **Object**
implements **Serializable**

	修饰符	类成员（节选）	功 能 说 明
1		**Number**()	构造方法
2		byte **byteValue**()	转换成 byte 类型
3		short **shortValue**()	转换成 short 类型
4		int **intValue**()	转换成 int 类型
5		long **longValue**()	转换成 long 类型
6		float **floatValue**()	转换成 float 类型
7		double **doubleValue**()	转换成 double 类型
		…	

本节习题

1. 下面的类(　　)不是 Java API 中的基本数据类型包装类。
   A. Byte　　B. Int　　C. Float　　D. Double
2. 字符类型 char 的包装类是(　　)。
   A. Char　　B. String　　C. Character　　D. character
3. 整数类 Integer 中将字符串转成 int 的方法是(　　)。
   A. intValue()　　B. toString()　　C. parseInt()　　D. valueOf()
4. 整数类 Integer 中将 int 转成字符串的方法是(　　)。
   A. toString()　　B. toString(int i)　　C. parseInt()　　D. valueOf()
5. 下面的类(　　)不是数值类 Number 的子类。
   A. Byte　　B. Boolean　　C. Float　　D. Double

5.4 Java 语言的根类 Object

在学习数学类 Math、字符串类 String、可变字符串类 StringBuilder、基本数据类型包装类和数值类 Number 的过程中，可发现这些类都直接或间接继承了同一个超类，这就是 Object 类。Object 到底是个什么类？

Object 类被称为**对象类**，它是 Java 语言的根类(root class)。

5.4.1 对象类 Object

所有 Java 语言中的类都直接或间接继承了对象类 Object。

- 如果定义类时没有继承超类,则默认继承 Object,这属于直接继承了 Object 类。
- 如果定义类时继承了某个超类,则属于间接继承 Object,因为所继承的超类也一定直接或间接继承了 Object 类。

Object 类可以使用的成员(public 或 protected 权限)都是方法,总共有 12 个。所有 Java 类都继承了这 12 个方法,定义类时可以重写这些方法。请读者仔细阅读下面的对象类 Object 说明文档。

java.lang.**Object** 类说明文档

public class **Object**

	修饰符	类成员(可以使用的全部成员)	功 能 说 明
1		**Object**()	构造方法
2		String **toString**()	将对象转换成字符串
3		Class <?> **getClass**()	取得当前运行类的对象
4		boolean **equals**(Object obj)	与另一个对象比较其内容是否相同
5		int **hashCode**()	将对象映射成一个哈希码(int 型整数)
6	protected	void **finalize**()	JVM 回收对象前自动调用该方法,其作用相当于 C++ 里的析构方法
7	protected	Object **clone**()	克隆一个当前对象并返回其引用
8		void **wait**()	当前线程进入阻塞状态,用于多线程编程
9		void **wait**(long timeout)	
10		void **wait**(long timeout, int nanos)	
11		void **notify**()	唤醒阻塞状态的线程,用于多线程编程
12		void **notifyAll**()	

例 5-5 给出一个对象类 Object 的测试程序示例代码。

例 5-5 一个对象类 Object 的测试程序示例代码(ObjectTest.java)

```
 1  public class ObjectTest {                              //测试类
 2      public static void main(String[ ] args) {          //主方法
 3          Object obj = new Object();
 4          System.out.println( obj );                     //显示引用变量: 类名@地址
 5          System.out.println( obj.toString() );          //转换成字符串
 6          Class<?> c = obj.getClass();                   //取得对象 obj 的运行类对象 c
 7          System.out.println( c.getName() );             //取得运行类对象的类名
 8          System.out.println( obj.getClass().getName() );    //直接取得对象 obj 的类名
 9          System.out.println();
10          //下面演示 4.1.1 节中的 Clock 类,该类直接继承了 Object 类
11          Clock cObj = new Clock(8, 30, 15);
12          System.out.println( cObj );                    //显示引用变量
13          System.out.println( cObj.toString() );   //转成字符串,可以重写 toString()方法
14          System.out.println( cObj.getClass().getName() );  //取得对象 cObj 的类名
15  }  }
```

在 Eclipse 集成开发环境中运行例 5-5 的程序,运行结果如图 5-7 所示。

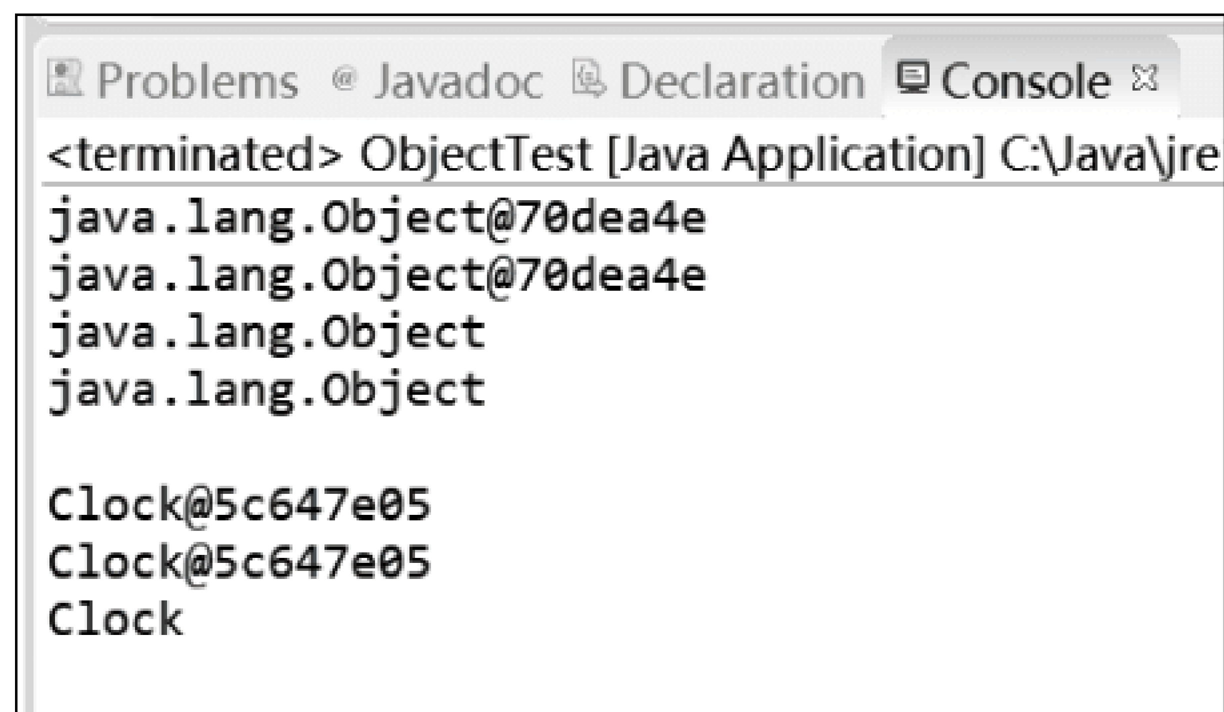


图 5-7 例 5-5 程序的运行结果

5.4.2 重写对象类 Object 的方法

所有 Java 语言(包括 Java API)中的类都直接或间接继承了对象类 Object。定义类时可以重写从 Object 继承来的方法。**请注意**,重写时,方法签名要与 Object 类里的方法签名完全相同。

(1) 重写方法 **toString**(),将对象转成可以理解的字符串。当显示对象或对象与字符串相加时,会自动调用 toString()方法将对象转成字符串。

(2) 重写方法 **hashCode**(),将对象映射成一个 int 型整数,今后可用于快速比较两个对象的内容是否相等。

(3) 重写方法 **equals**(),比较两个对象的内容是否相等。**注**:关系运算符"=="只能比较两个引用是否相等,即是否引用了同一个对象。

(4) 重写方法 **clone**(),创建一个和当前对象内容一样的新对象(即克隆),并返回其引用。**注**:重写方法 clone()的类需实现可克隆的接口 Cloneable,否则 clone()方法没有激活,不能使用。

(5) 重写方法 **finalize**(),完成对象回收之前的善后工作,例如将对象数据保存到硬盘文件。Java 虚拟机在回收对象前会自动调用其所属类的 finalize()方法。**注**:finalize()方法的语法作用相当于 C++语言里的析构方法,Java 语言没有析构方法。

例 5-6 给出一个重写 Object 方法的新钟表类 Clock 及其测试类的示例代码。**注**:原钟表类 Clock 请参见 4.1.1 节中的例 4-1。

例 5-6 一个重写 Object 方法的新钟表类 Clock 及其测试类的示例代码

```
1  class Clock implements Cloneable {   //自动继承 Object 类,实现接口 Cloneable(Clock.java)
2      //此处省略例 4-1 中类 Clock 已有的代码,下面演示重写从 Object 类继承来的方法
3      public String toString()          //重写方法 toString()
4      {  return String.format("Clock@%d:%d:%d", hour, minute, second);  }
5      public int hashCode()             //重写方法 hashCode()
6      {  return second;  }              //生成哈希码:简单地将秒数作为钟表对象的哈希码
7      public boolean equals(Object obj) { //重写方法 equals()
8          if ((obj instanceof Clock) == false)  return false;     //类型不同,则直接返回
9          Clock c = (Clock)obj;         //将 Object 类型转换成 Clock 类型
```

```
10          return c.hour  ==  hour && c.minute  ==  minute && c.second  ==  second;
11          //比较两个钟表对象的时间,如果时、分、秒都相同则返回 true,否则返回 false
12      }
13      public Object clone() throws CloneNotSupportedException      //重写方法 clone ()
14      {  Clock c  =  (Clock)super.clone();   return c;   }           //克隆一个钟表对象
15      //注: clone ()方法头后面的"throws …"是 Java 语言的异常处理,将在后面讲解
16  }
```

```
1  public class ObjectTest {                         //测试类(ObjectTest.java)
2      public static void main(String[] args) {//主方法
3          Clock cObj  =  new Clock(8, 30, 15);
4          System.out.println( cObj );//显示引用变量
5          System.out.println( cObj.toString() );//转换成字符串,使用重写的 toString()
6          System.out.println( cObj.getClass().getName() );     //取得对象 cObj 的类名
7          //下面演示如何比较两个钟表对象是否相等
8          Clock cObj1  =  new Clock(8, 30, 15);//新建对象,设置与 cObj 相同的时间
9          Clock cObj2  =  cObj;                      //cObj2 与 cObj 引用同一钟表对象
10         System.out.println( cObj1  ==  cObj );  //检查引用是否相同,即是否引用了同一对象
11         System.out.println( cObj2  ==  cObj );  //检查两个引用是否相同
12         System.out.println( cObj1.equals(cObj) );  //检查两个对象的内容(时间)是否相同
13         System.out.println( cObj.hashCode() );     //显示 cObj 对象的哈希码
14         System.out.println( cObj1.hashCode() );    //显示 cObj1 对象的哈希码
15         //下面演示如何克隆一个钟表对象
16         try {   //try - catch 是 Java 语言的异常处理,将在后面讲解
17             Clock cObj3  =  (Clock)cObj.clone();//克隆一个和 cObj 一样的对象
18              System.out.println( cObj3.toString() );      //检查克隆对象的内容是否相同
19         } catch(CloneNotSupportedException e) { };
20 }  }
```

在 Eclipse 集成开发环境中运行例 5-6 的测试程序 ObjectTest.java,运行结果如图 5-8 所示。**注:** 如果将例 5-6 与例 5-5 放在同一个目录下(即同一个包中),则两个测试类 ObjectTest 会出现重名的问题,可以将其中一个更名为 ObjectTest1。

```
Problems  Javadoc  Declaration  Console
<terminated> ObjectTest [Java Application] C:\Java\jre
Clock@8:30:15
Clock@8:30:15
Clock
false
true
true
15
15
Clock@8:30:15
```

图 5-8 例 5-6 中测试程序 ObjectTest.java 的运行结果

需要特别说明的是,如果重写类的 equals()方法,则应当同时重写 hashCode()方法和 toString()方法。因为如果两个对象内容相等,即 equals()返回 true,则 hashCode()方法所返回的哈希码应当一样,toString()方法所转换出的字符串也应当一样。这就需要程序员在

重写 equals()方法后,必须同步重写 hashCode()方法和 toString()方法。

例 5-6 在定义钟表类 Clock 时还实现了一个可克隆的接口 Cloneable。接口 Cloneable 未定义任何成员,是一个空接口。其定义代码如下:

```
public interface Cloneable;            //接口 Cloneable 的定义代码
```

这种未定义任何成员的空接口称作**标记接口**(marker interface)。定义类时实现某个标记接口,其语法作用是为类激活(或称启用)某种功能。例如,钟表类 Clock 实现接口 Cloneable 的目的是为了激活克隆功能。类只有在激活克隆功能后,调用其中的 clone()方法才能真正实现克隆的功能。

标记接口未定义任何抽象方法,因此实现时也不需要编写任何具体的算法代码。

5.4.3 已探索的 Java API 类库

截至目前,已学习了数学类 Math、字符串类 String、可变字符串类 StringBuilder、基本数据类型包装类、数值类 Number,还有 Java 语言的根类 Object。图 5-9 给出了这些类的继承关系和接口实现示意图。

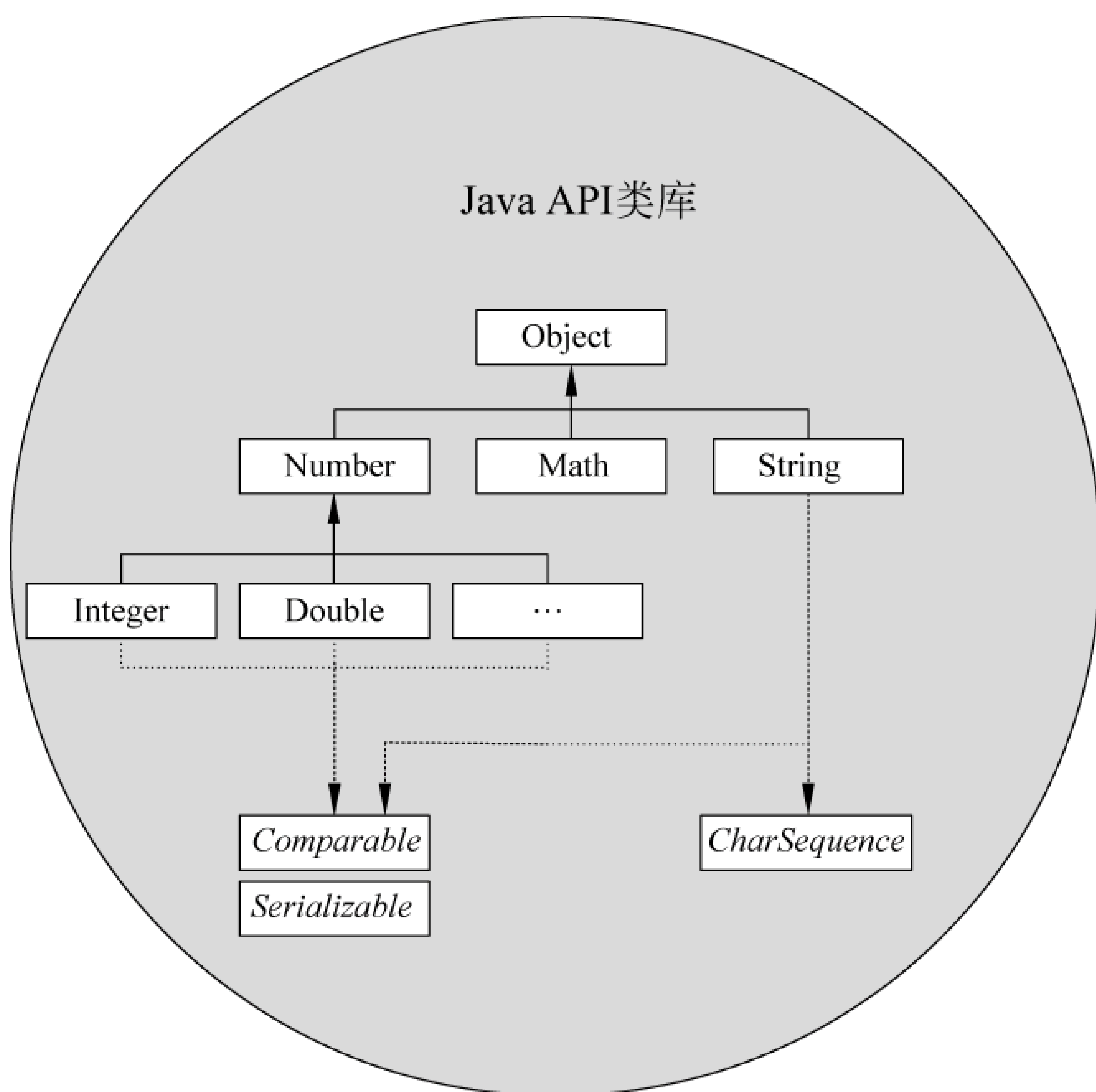


图 5-9 已探索的 Java API 类库

学习 Java API,要了解其中每个类的继承关系和所实现的接口,了解有哪些类族和接口族。这一点很重要,因为只有同一类族或同一接口族中的类才能共用算法代码。换句话说,利用对象的替换与多态机制,处理超类或接口的算法可以用于处理其所有子类的对象。

Java API 大量运用类的继承、接口的实现,以及对象的替换与多态机制,最大程度上实

现了代码的重用或共用。

本节习题

1. 对象类 Object 中将对象转成字符串的方法是(　　)。
   A. toString()　　B. equals()　　C. hashCode()　　D. finalize()
2. 对象类 Object 中比较两个对象内容相等的方法是(　　)。
   A. toString()　　B. equals()　　C. hashCode()　　D. finalize()
3. Java 虚拟机在回收对象之前会自动调用对象的方法成员(　　)。
   A. toString()　　B. equals()　　C. hashCode()　　D. finalize()
4. Java 语言中所有类都包含的成员是(　　)。
   A. toString()　　B. compareTo()　　C. length()　　D. valueOf()
5. 处理 Object 类对象的算法代码不能用于处理(　　)类型的数据。
   A. String　　B. StringBuilder　　C. Integer　　D. int

5.5 系统类 System

请读者阅读下面的系统类 System 说明文档,其中主要包含一些静态字段和静态方法。

java. lang. **System** 类说明文档

public final class **System**
extends **Object**

	修饰符	类成员(节选)	功 能 说 明
1	static final	InputStream **in**	字段,标准输入流对象
2	static final	PrintStream **out**	字段,标准输出流对象
3	static final	PrintStream **err**	字段,标准错误输出流对象
4	static	String **getProperty**(String key)	读取计算机系统属性
5	static	String **setProperty**(String key, String value)	设置计算机系统属性
6	static	void **arraycopy**(Object src, int srcPos, Object dest, int destPos, int length)	复制数组
7	static	long **currentTimeMillis**()	读取系统时间
8	static	void **gc**()	请求回收垃圾
9	static	void **exit**(int status)	退出当前程序的运行
10	static	void **loadLibrary**(String libname)	加载本地库文件

…

下面简要介绍一下系统类 System 的主要功能。

1. 输入和输出

系统类 System 定义了 3 个静态字段,分别是标准输入流对象 in、标准输出流对象 out

和标准错误输出流对象 err。例如：

```
System.out.println( "Hello, World" );          //在显示器上显示"Hello, World"
```

其中：

- **System**：这是系统类的类名。
- **out**：这是系统类 System 包含的字段成员名。out 是输出流类 PrintStream 的对象。
- **println**：这是对象 out 包含的下级方法成员名，其功能是在显示器上显示信息。

注：System.in、System.out、System.err 相当于 C++语言里的 cin、cout 和 cerr。

2. 读取或设置计算机系统属性

表 5-2 给出了一台计算机系统所具有的主要属性(property)。Java API 分别用字符串形式的键(key)来指代不同的属性。

表 5-2 计算机系统的主要属性

属性的键	说明
"java.class.path"	Java 类库的搜索路径
"java.home"	Java 运行环境(JRE)的安装目录
"java.version"	JRE 版本号
"os.arch"	操作系统架构
"os.name"	操作系统名称
"os.version"	操作系统版本号
"user.dir"	用户工作目录(当前目录)
"user.home"	用户根目录
"user.name"	用户账号

使用系统类 System 中的静态方法 getProperty()、setProperty()就可以读取或设置当前计算机系统的相关属性。例如：

```
System.out.println( System.getProperty("os.name") );     //读取并显示当前操作系统的名称
```

3. 复制数组

复制数组时，程序员通常需要使用循环语句遍历数组，逐个复制数组里的所有元素。使用系统类 System 中的静态方法 arraycopy()可以很方便地复制数组。例如：

```
int x[ ] = { 1, 2, 3, 4, 5 };
int y[ ] = new int[3];
// y = x;                                   //错误:不能实现复制数组的功能
System.arraycopy(x, 1, y, 0, 3);            //从 x[1]开始将元素复制给 y[0],共复制 3 个元素
System.out.println(y[0] +"," +y[1] +"," +y[2]); //显示复制后的结果:2,3,4
```

4. 读取系统时间

调用系统类 System 中的静态方法 currentTimeMillis()可以读取当前计算机系统的时

间。这个时间是自 1970 年 1 月 1 日零时起,至系统当前时刻的毫秒数。例如:

```
System.out.println( System.currentTimeMillis() );    //读取并显示当前计算机系统的时间
```

5. 请求回收垃圾

调用系统类 System 中的静态方法 gc(),可以请求 Java 虚拟机的垃圾回收器(garbage collector)回收系统中当前未被引用的对象的内存单元。未被引用的对象,应该是程序已经不再使用的对象,其内存单元可以被收回。例如:

```
System.gc();                                          //请求回收垃圾
```

6. 退出当前程序

调用系统类 System 中的静态方法 exit()可以退出当前程序,同时也停止当前 Java 虚拟机的运行。

```
System.exit( 0 );                                     //退出当前程序的运行
```

本节习题

1. 系统类 System 定义了几个输入输出流对象字段,其中不包括(　　)。
   A. in　　B. out　　C. err　　D. log
2. 系统类 System 中读取计算机系统属性的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()
3. 系统类 System 中复制数组的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()
4. 系统类 System 中读取系统时间的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()
5. 系统类 System 中请求 Java 虚拟机回收垃圾的方法是(　　)。
   A. getProperty()　　B. arraycopy()
   C. currentTimeMillis()　　D. gc()

5.6 异常处理

程序中的错误可分为 3 种,分别是**语法**(syntax)错误、**语义**(semantics)错误(或称为**逻辑**错误),以及**运行时**(runtime)错误。针对不同错误,Java 语言具有不同的解决办法,最终保证所开发的程序能够正确、稳定地运行。

Java 语言针对程序运行时错误设计了专门的**异常处理**机制,即 try-catch 机制。Java

API 为异常处理机制提供描述不同异常情况的**异常类**。

5.6.1 3 种不同的程序错误

本节通过具体的程序实例，分别讲解什么是程序中的语法错误、语义错误和运行时错误，以及它们对应的解决办法。

1. 语法错误

例 5-7 给出一个简单的 Java 除法运算程序，其中存在语法错误。

例 5-7 一个简单的 Java 除法运算程序(存在**语法错误**)(SyntaxError.java)

```
 1  import java.util.Scanner;
 2  public class SyntaxError {                        //一个存在语法错误的类
 3      int Div(int n) {                              //方法功能:求 100÷n
 4          int result;
 5          result = 100 / n;                         //求 100÷n
 6          return result;
 7      }
 8
 9      public static void main(String[] args) {      //主方法是一个静态方法
10          int N;
11          Scanner sc = new Scanner( System.in );    //创建键盘扫描器对象
12          N = sc.nextInt();                         //键盘输入 N 的值
13          int retValue = Div( N );                  //语法错误:调用非静态方法 Div 计算 100÷N
14          System.out.println( "100÷" + N + " = " + retValue );
15  } }
```

例 5-7 中，代码第 13 行有一个语法错误：静态的 main()方法不能调用非静态的 Div()方法。在 Eclipse 集成开发环境中运行该程序，编译器会提示如下错误信息：

```
Cannot make a static reference to the non - static method Div(int) from the type ErrorSyntax.
```

程序员应按照提示信息，查找错误原因并修改程序。按如下形式将例 5-7 中的方法 Div()定义为静态方法(代码第 3 行)：

```
static  int Div(int n) {              //方法功能:求 100÷n
```

这样就完成了语法错误的修改。再次运行修改后的程序，没有任何语法错误，编译通过。

如果程序员未能严格按照语法规则编写程序，这就属于**语法**错误。编译时，Java 编译器负责检查源程序中的语法错误，如无语法错误则将其编译成字节码程序，否则将提示错误信息。Java 编译器能够帮助程序员检查出所有的语法错误，因此语法错误易于检查，易于修改。

2. 语义错误

例 5-8 给出另一个 Java 除法运算程序，其中存在**语义错误**。代码第 5 行本应是除法运

算,但被错误写成了乘法运算,语法正确但语义错误。

例 5-8 另一个简单的 Java 除法运算程序(存在**语义错误**)(SemanticsError. java)

```
1  import java.util.Scanner;
2  public class SemanticsError {       //一个存在语义错误的类
3      static int Div(int n) {         //方法功能:求 100 ÷ n
4          int result;
5          result = 100 * n;           //语义错误:将除法错误写成了乘法,语法正确但语义错误
6          return result;
7      }
8
9      public static void main(String[ ] args) {              //主方法
10         int N;
11         Scanner sc = new Scanner( System.in );             //创建键盘扫描器对象
12         N = sc.nextInt();                                  //键盘输入 N 的值
13         int retValue = Div( N );                           //调用方法 Div 计算:100 ÷ N
14         System.out.println( "100 ÷ " + N + " = " + retValue );
15 } }
```

在 Eclipse 集成开发环境中运行例 5-8 的程序,无任何语法错误,可以正常运行。例如,输入 2:

```
2 <回车键>
```

程序将显示如下结果:

```
100 ÷ 2 = 200
```

这个结果显然是错误的,正确结果应为:

```
100 ÷ 2 = 50
```

运行测试的结果表明,程序中存在**语义**错误,即程序的算法逻辑有错误。程序员需检查源程序,找出错误原因。本例中,将代码第 5 行的"100 * n"改成"100 / n",这样就完成了对语义错误的修改。

Java 编译器不能帮助程序员发现语义错误。程序员必须通过运行测试,比对程序结果才能发现语义错误。

3. 运行时错误

同样的除法运算,例 5-9 给出一份既没有语法错误,也没有语义错误的 Java 程序代码。

例 5-9 一个无任何语法或语义错误的 Java 除法运算程序(NoError. java)

```
1  import java.util.Scanner;
2  public class NoError {                    //一个无任何语法或语义错误的类
3      static int Div(int n) {               //方法功能:求 100 ÷ n
4          int result;
5          result = 100 / n;                 //求 100 ÷ n
```

```
 6            return result;
 7        }
 8
 9        public static void main(String[] args) {        //主方法
10            int N;
11            Scanner sc = new Scanner( System.in );    //创建键盘扫描器对象
12            N = sc.nextInt();                         //键盘输入 N 的值
13            int retValue = Div( N );                  //调用方法 Div 计算:100÷N
14            System.out.println( "100÷" + N + " = " + retValue );
15    } }
```

在 Eclipse 集成开发环境中运行例 5-9 的程序，没有语法错误，可以正常运行。例如，输入 2：

```
2 <回车键>
```

程序将显示如下结果：

```
100÷2 = 50
```

运行结果也正确。但再次运行该程序，输入 0：

```
0 <回车键>
```

这时 Eclipse 将提示该程序运行出现了一个算术运算异常，如图 5-10 所示。因为任何数都不能被零除，计算机无法执行“100/0”这样的运算，因此将停止程序的运行。

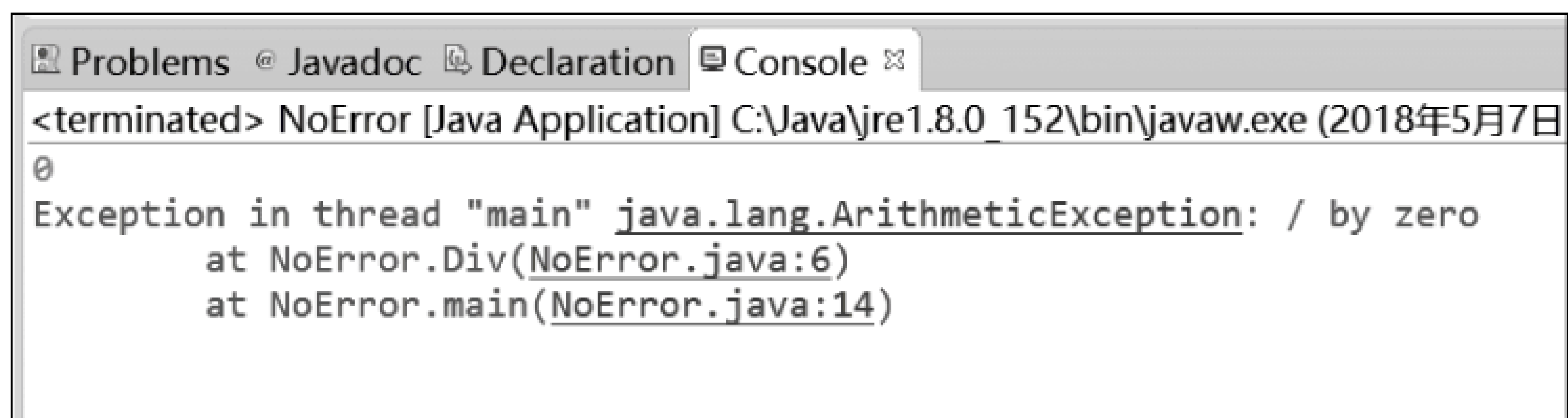


图 5-10　运行例 5-9 程序时出现的“被零除”异常

程序运行时，因运行环境差异或用户操作不当所造成的程序错误统称为**运行时**错误。运行环境存在差异，或用户出现操作不当，这些都称为程序运行时的**异常**(exception)。程序员应当对程序运行时可能出现的异常情况进行处理。

5.6.2　Java 语言的异常处理机制

程序运行过程中常见的异常情况如下。

- 用户操作不当。例如，运行例 5-9 的除法运算程序时输入了 0。
- 输入文件不存在。从文件输入数据，但文件不存在，这会导致文件输入异常。
- 网络连接中断。程序可以通过网络发送、接收数据，网络连接中断将导致网络通信异常。
- 非法访问内存单元。数组越界或空引用会导致程序非法访问内存单元异常。

程序员在编写程序时,应该能够预见到程序运行时可能会发生哪些异常,并在程序中添加异常处理机制,避免因异常情况而导致程序死机或意外中断等严重错误。

1. Java语言的异常处理机制

Java语言中,一个**异常处理机制**由如下3部分组成。

(1) **发现异常**。程序员应在可能出现异常的程序位置增加检查异常的代码,其目的是及时发现异常。Java语言使用 **if** 语句来检查异常。Java API为描述不同的异常情况专门提供了一组**异常类**。

(2) **报告异常**。发现异常后,程序应向Java虚拟机报告异常。Java语言使用 **throw** 语句来报告异常。

(3) **处理异常**。Java虚拟机在接收到异常报告后,将立即改变程序原来的执行流程,跳转去执行异常处理代码。所谓异常处理,就是在程序算法中增加异常处理流程。没有异常时,程序执行正常算法流程;发现异常时,程序执行异常处理流程。Java语言使用 **try-catch** 语句来编写捕获和处理异常的代码。

为例5-9的除法运算程序添加Java异常处理机制,具体代码如例5-10所示。假设程序要求键盘输入的数必须为正整数,输入0或负数为异常情况。

例 5-10 一个添加了异常处理机制的Java除法运算程序(ErrorTryCatch.java)

```
1   import java.util.Scanner;
2   public class ErrorTryCatch {                      //一个添加了异常处理机制的类
3       static int Div(int n) {                       //方法功能:求 100 ÷ n
4           int result;
5           if (n <= 0)                               //检查异常:如果 n<=0,则属于异常情况
6               throw ( new RuntimeException("输入的数值必须为正整数") );  //报告异常
7           result = 100 / n;
8           return result;
9       }
10
11      public static void main(String[] args) {      //主方法
12          int N;
13          Scanner sc = new Scanner( System.in );//创建键盘扫描器对象
14          N = sc.nextInt();                         //键盘输入 N 的值
15          try {                                     //启用 Java 异常处理机制
16              int retValue = Div( N );              //调用方法 Div 计算:100 ÷ N
17              System.out.println( "100 ÷ " + N + " = " + retValue );
18          }
19          catch( RuntimeException e)                //捕获并处理异常
20          {  System.out.println( e.getMessage() );  }
21  }  }
```

在Eclipse集成开发环境中运行例5-10的程序,输入0:

```
0 <回车键>
```

这时 Eclipse 不会像例 5-9 那样意外中断程序执行，而是按照程序所设计的异常处理流程向用户显示一条提示信息，如图 5-11 所示。

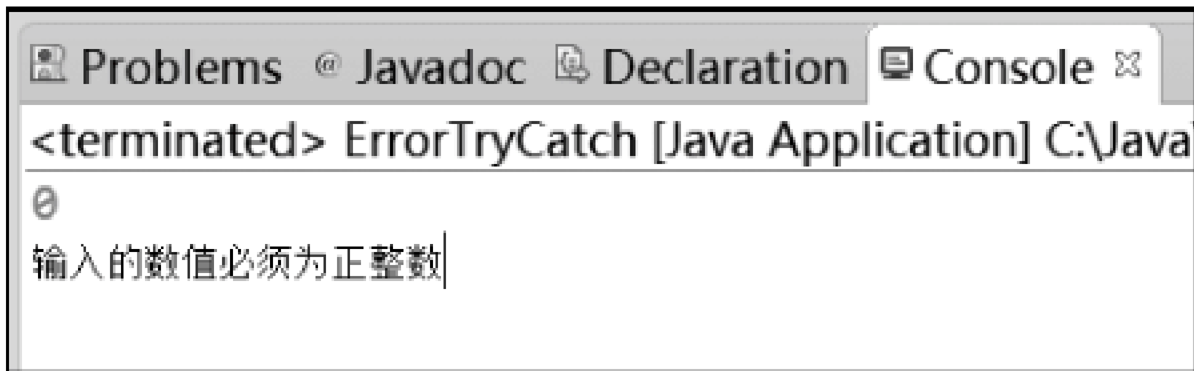


图 5-11　例 5-10 程序能够捕获并处理“被零除”的异常

2. Java API 提供的异常类族

Java API 总结了各种可能的异常情况，然后将它们定义成异常类，其中包括描述异常的相关信息。异常类是一个类族(见图 5-12)，其根类是 **Throwable**(可抛出的类)。表 5-3 列出了这个类族中比较常用的异常类，并给出简要的功能说明。

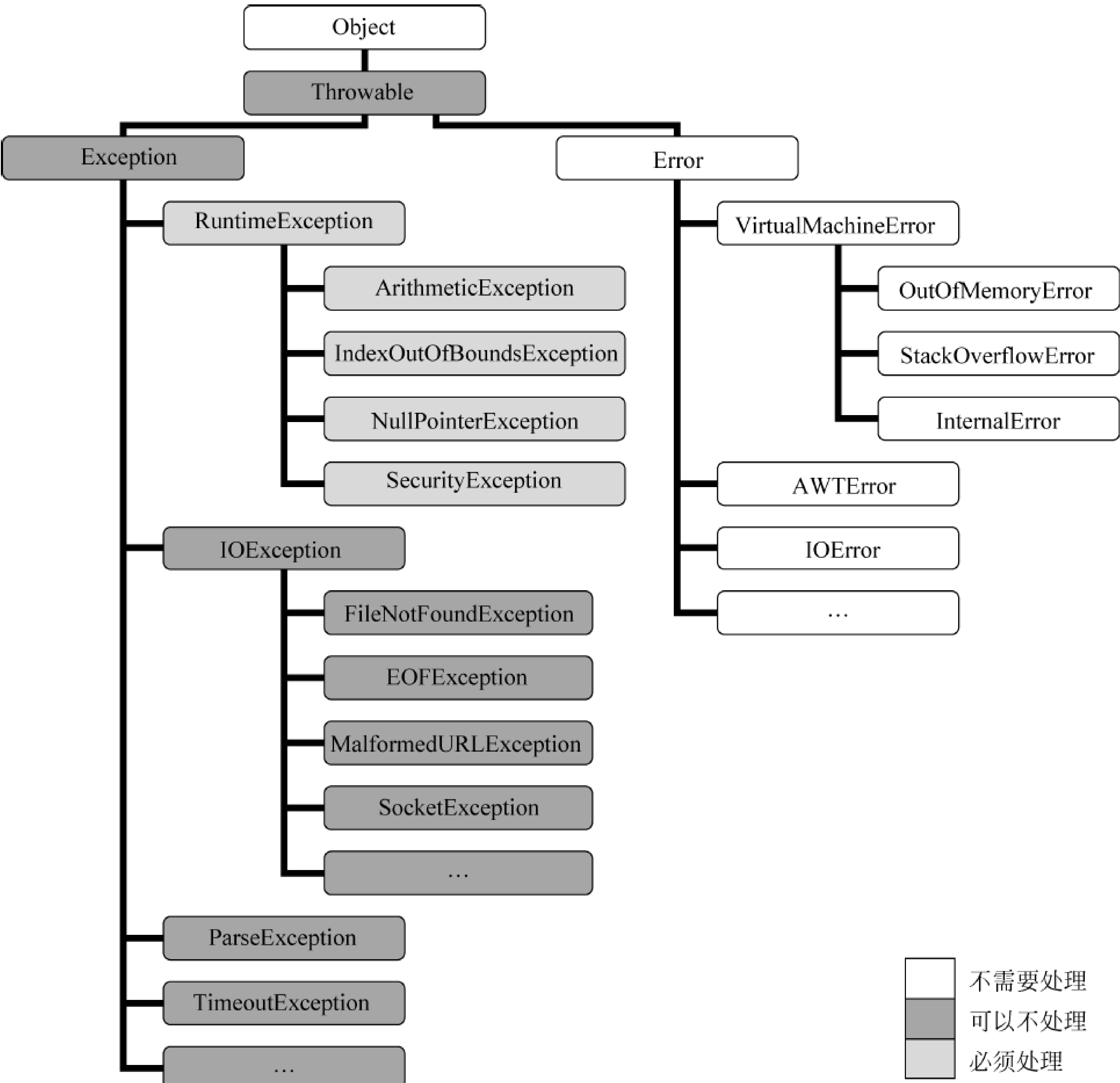


图 5-12　Java API 提供的异常类族

表 5-3 常用的异常类

异 常 类	功 能 说 明
Error	错误类
Exception	异常类
RuntimeException	运行时异常类
ArithmeticException	算术异常类
IndexOutOfBoundsException	下标越界异常类
NullPointerException	空指针(空引用)异常类
SecurityException	安全性异常类
IOException	I/O 读写异常类
FileNotFoundException	未找到文件异常类
EOFException	文件结束异常类
MalformedURLException	URL 格式异常类
SocketException	Socket 连接异常类
ParseException	字符串解析异常类
TimeoutException	超时异常类

请读者阅读下面的异常根类 Throwable 说明文档。

java. lang. **Throwable** 类说明文档

public class **Throwable**
extends **Object**
implements **Serializable**

	修 饰 符	类成员(节选)	功 能 说 明
1		**Throwable**()	构造方法
2		**Throwable**(String message)	构造方法
3		**Throwable**(String message, Throwable cause)	构造方法
4		String **getMessage**()	获取错误信息
5		Throwable **getCause**()	获取错误原因
6		void **printStackTrace**()	打印错误的轨迹

…

3. throw 语句

Java 语言使用 throw 语句向 Java 虚拟机抛出一个异常对象,其目的是向 Java 虚拟机报告异常。异常对象必须是属于 Throwable 类族中异常类的对象,不能是任何其他类的对象。异常对象中包含了描述异常的相关信息。throw 语句有 3 种常用句型。

1) 基本句型

```
异常类名 eRef = new 异常类名( "异常信息" );      //先创建异常对象
throw  eRef;                                   //然后抛出异常对象
```

2) 简写句型

```
throw  new 异常类名( "异常信息" );              //创建异常对象并立即抛出
```

3）链接句型

```
throw  eRefLast;              //接力抛出已被捕获的异常对象 eRefLast,形成异常处理链条
```

或

```
throw  new 异常类名( "附加异常信息", eRefLast); //抛出新异常对象,形成异常处理链条
```

计算机执行 throw 语句，会在抛出异常对象后立即改变执行流程，跳转去执行异常处理代码。程序员使用 try-catch 语句来编写捕获和处理异常的程序代码。

4. try-catch 语句

Java 语法：**try-catch 语句**

```
try {
  受保护代码(其中可能会直接或间接抛出异常)
}
catch ( 异常类型 1 引用变量 )
{  异常类型 1 的处理代码 }
catch ( 异常类型 2 引用变量 )
{  异常类型 2 的处理代码 }
…
finally
{  最终的善后处理代码 }
```

语法说明：

- **try-catch-finally** 语句是一个整体，其中包含 try 子句、catch 子句和 finally 子句。try 子句后面至少跟一条 catch 子句，或 finally 子句。finally 子句是可选项。
- **try** 子句：如果预计某个程序代码段在执行时可能发生异常，则程序员可使用 try 子句将该代码段保护起来。Java 虚拟机在执行受保护代码段时将启用异常处理机制，监控代码执行过程中的任何异常报告，包括代码所调用下级方法的异常报告。
- **catch** 子句：catch 子句负责捕获并处理异常，每个 catch 子句只负责一种类型的异常。
  - 若受保护代码段在执行过程中发生异常，抛出了某个异常对象，则 Java 虚拟机会根据异常类型依次匹配 catch 子句。如果异常对象的类型与某个 catch 子句中的异常类型匹配，称异常对象被**捕获**。此时 catch 子句中的引用变量将引用被捕获的异常对象。需要注意的是，超类可以匹配子类，即捕获超类的 catch 子句将会同时捕获到其所有子类的异常，因此通常将捕获超类的 catch 子句放在捕获子类 catch 子句的后面。
  - 如果异常对象被某个 catch 子句捕获，则执行该 catch 子句的处理代码。处理代码负责对异常情况进行处理，例如向用户显示提示信息；也可以使用 throw 语句的链接句型接力抛出异常，形成一个异常处理链条。可以调用异常类的方法 printStackTrace()显示异常处理链条的轨迹。
  - 每个异常最多只会被一个 catch 子句捕获，因此只会有一个 catch 子句的处理代码被执行，其他 catch 子句都不会被执行。

- 如果异常未被任何 catch 子句捕获,Java 虚拟机会自动逐级交由上级方法捕获、处理,直到被上级方法中的某个 catch 子句捕获。如果连最上级的主方法 main()也未能捕获异常,则中止当前程序的执行,并显示相关的异常信息。
- 若受保护代码段在执行过程中未抛出任何异常对象,即没有发生异常,则所有的 catch 子句都不会被执行。

■ **finally** 子句:finally 子句为正常处理流程和异常处理流程提供统一的善后处理。简单地说,不管是否发生了异常,finally 子句中的代码都会被执行。finally 子句通常用于清理程序所占用的资源,例如关闭已打开的文件。

例 5-11 给出了一个 Java 异常处理机制的演示程序。

例 5-11 一个 Java 异常处理机制的演示程序(ExceptionTest.java)

```
1   import java.io.IOException;
2   public class ExceptionTest {                              //测试类
3       static void fun(int choice) {     //根据参数 choice 模拟不同的异常,然后进行异常处理
4           System.out.println( "choice: " + choice ); //显示提示信息,用于观察执行流程
5           System.out.println( "Before try - catch" );
6
7           try {
8               System.out.println( "Before throw" );
9               if (choice == 1)                              //1:模拟算术运算异常
10                  throw new ArithmeticException("ArithmeticException");
11              else if (choice == 2)                         //2:模拟输入输出异常
12                  throw new IOException("IOException");
13              System.out.println( "After throw" );
14          }
15          catch( ArithmeticException e)                     //捕获并处理算术运算异常
16          {   System.out.println( e.toString() );   }
17          catch( IOException e)                             //捕获并处理输入输出异常
18          {   System.out.println( e.toString() );   }
19          finally                                           //善后处理
20          {   System.out.println( "finally block" );   }
21
22          System.out.println( "After try - catch" );
23      }
24
25      public static void main(String[ ] args)               //主方法
26      {   fun( 1 );   }          //通过不同实参来模拟不同的异常,例如 fun(1)、fun(2)、fun(0)
27  }
```

例 5-11 中,主方法 main()在调用子方法 fun()时通过不同实参来模拟不同的异常。在 Eclipse 集成开发环境中运行这个程序,对比屏幕提示信息(见图 5-13)和源代码,观察程序的执行流程,这样可以深入理解 Java 异常处理机制的工作原理。

```
Problems  Javadoc  Declaration  Console
<terminated> ThrowTest [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe
choice: 1
Before try-catch
Before throw
java.lang.ArithmeticException: ArithmeticException
finally block
After try-catch
```

(a) 主方法调用fun(1)模拟算术运算异常

```
Problems  Javadoc  Declaration  Console
<terminated> ThrowTest [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe
choice: 2
Before try-catch
Before throw
java.io.IOException: IOException
finally block
After try-catch
```

(b) 主方法调用fun(2)模拟输入输出异常

```
Problems  Javadoc  Declaration  Console
<terminated> ThrowTest [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe
choice: 0
Before try-catch
Before throw
After throw
finally block
After try-catch
```

(c) 主方法调用fun(0)模拟不发生异常的情况

图 5-13 例 5-11 程序的运行结果

5.6.3 Java 异常处理的代码框架

Java 程序通常都存在多级嵌套调用关系。例如，程序的主方法 main()调用子方法 fun1()，fun1()再调用子方法 fun2()，……，最终可能会调用 Java API 中的方法(见图 5-14)。其中，主方法 main()位于最顶层，Java API 则处于最底层。

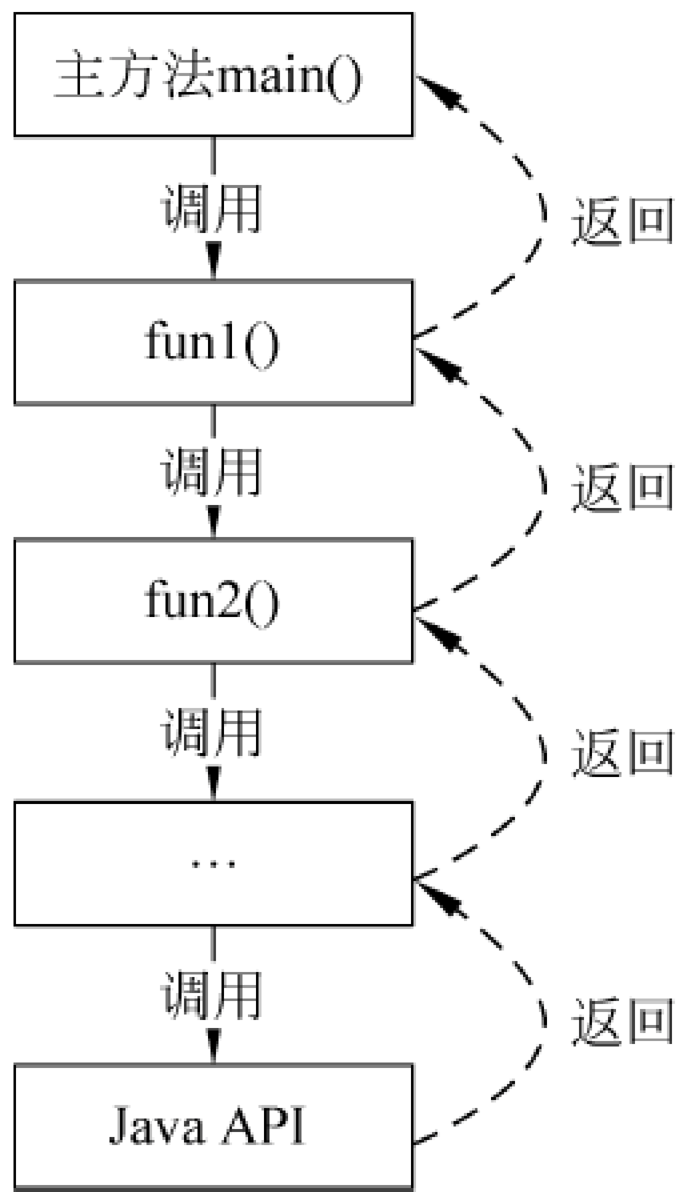


图 5-14 方法的多级嵌套调用

假设图 5-14 中的方法 fun2()在执行过程中可能会发生异常，该如何处理这个异常呢？程序员可以按如下 3 种不同的思路来设计异常处理流程。

1. 由方法 fun2()自己处理

如果由方法 fun2()自己处理所报告的异常，则程序员应当按如下代码框架来定义方法 fun2()。

```
… fun2() {          //方法 fun2()处理自己异常时的代码框架
  …
   try {            //启用异常处理机制
```

```
        …
        if (发现异常)   throw 异常对象;     //报告异常
        …
    }
    catch ( … )  {  异常处理代码 }          //捕获并处理异常
    …
}
```

2. 将异常交由上级方法 fun1()处理

方法 fun2()也可以只报告异常,将异常处理的工作交由其上级方法 fun1()去完成。这时,程序员应当按如下代码框架来定义方法 fun2()和 fun1()。

```
… fun2() {                          //方法 fun2()只报告但不处理异常时的代码框架
    …
    if (发现异常)   throw 异常对象;     //报告异常
    …
}

… fun1() {                          //方法 fun1()处理下级方法 fun2()所报告异常时的代码框架
    …
    try {                           //启用异常处理机制
        …
        fun2();                     //调用方法 fun2(),调用过程中可能会报告异常
        …
    }
    catch ( … )  {  异常处理代码 }     //捕获并处理异常
    …
}
```

下级方法在发现异常时只报告(即抛出异常对象)但不处理,由上级方法统一捕捉、处理所有的异常,这就是 Java 程序中的**多级异常处理**。

多级异常处理可以将分散在不同方法中的异常处理代码剥离出来,集中交由某个上级方法统一进行处理。多级异常处理的好处主要体现在以下两个方面。

- 将原来分散在不同方法中的异常处理代码集中到一起,这样可以减少异常处理中重复的代码。
- 将方法中的异常处理代码剥离出来,让方法专注于正常算法流程,这样可以简化算法设计,优化代码结构。

3. 多级异常处理链条

方法 fun1()在处理完 fun2()所报告的异常之后,可以向更上级的方法继续报告异常。在图 5-14 中,这个更上级的方法就是主方法 main()。这时,程序员应当按如下代码框架来修改 fun1(),并在主方法 main()中添加异常处理代码。

```
… fun1() {     //方法 fun1()在处理下级方法 fun2()报告的异常后,继续向更上级的方法报告异常
    …
    try {                       //启用异常处理机制
```

```
        …
        fun2();              //调用方法 fun2(),调用过程中可能会报告异常
        …
    }
    catch ( … e ) {          //第 1 次捕获异常对象 e
        异常处理代码 1;        //第 1 次处理异常对象 e
        throw  e;            //接力抛出已被捕获的异常对象 e,形成异常处理链条
        //或: throw  new 异常类名( "附加异常信息", e);  //接力抛出新的异常对象
    }
    …
}

… main( … ) {               //主方法 main()处理下级方法 fun1()所报告异常时的代码框架
    …
    try {                    //启用异常处理机制
        …
        fun1();              //调用方法 fun1()
        …
    }
    catch ( … e ) {          //第 2 次捕获异常对象 e
        异常处理代码 2;        //第 2 次处理异常对象 e
    }
    …
}
```

捕获异常对象,处理后再接力抛出,交由更上级的方法继续捕捉、处理,这就形成了一个多级异常处理链条。这么做的原因是,某些异常需要经过不同层级代码多次处理才能完成,每个层级只是整个异常处理流程中的一个环节。

5.6.4 不同性质的异常

综合分析 Java 程序运行过程中可能发生的各种异常,可以按发生原因将它们划分成 3 种不同的性质,分别是系统导致的异常、程序员导致的异常,还有用户导致的异常。针对这 3 种不同性质的异常,Java 语言要求程序员区别对待,分别按不同的方式去处理它们。

1. 3 种不同性质的异常

1) 系统异常

由于 Java 虚拟机或 Java API 自身原因导致的异常称为系统异常。Java API 将描述系统异常的类都定义成 **Error** 类的子类,可统称为系统异常类。例如,描述 Java 虚拟机内存不足的异常类 OutOfMemoryError、描述 Java 虚拟机内部错误的异常类 InternalError 等,参见图 5-12。

程序员无法预见系统异常,也处理不了系统异常。因此 Java 语言在语法上不强制要求程序员必须使用异常处理机制去捕获或处理系统异常。针对系统异常,程序员可以处理,也可以不做任何处理。如果程序运行过程中发生了系统异常,唯一能做的只能是中断程序的执行。

2）编程异常

因程序员在编程时考虑不周而导致的异常称为编程异常。Java API 将描述编程异常的类都定义成 **RuntimeException** 类的子类，可统称为编程异常类。例如，描述被零除等算术运算错误的异常类 ArithmeticException、描述空引用访问错误的异常类 NullPointerException、描述数组越界错误的异常类 IndexOutOfBoundsException 等，参见图 5-12。

如果程序运行过程中因编程异常导致程序意外中断，这对用户来说是不可理解的，也是不可接受的。Java 语言认为，程序员应当通过周密的设计和完善的测试，完全杜绝程序中的编程异常，否则就是失职。因此对于编程异常，Java 语言在语法上也不强制要求程序员必须使用异常处理机制进行处理，即程序员可以处理，也可以不做任何处理。

3）用户异常

因操作不当或计算机系统配置不当等用户因素而导致的异常称为用户异常。Java API 将描述用户异常的类都定义成 **Exception** 类下除 RuntimeException 之外的其他子类，它们可统称为用户异常类。例如，描述输入输出错误的异常类 IOException、描述未找到输入文件错误的异常类 FileNotFoundException、描述网路连接错误的异常类 SocketException 等，参见图 5-12。

程序运行过程中，如果发现操作不当等用户异常，Java 程序不应该中断执行，而应该立即捕捉异常并向用户显示错误提示，同时还应保持程序正常运行，让用户能够继续操作程序。为了保证这一点，Java 语言在语法上强制要求程序员必须添加异常处理机制对用户异常进行处理。如果程序员没有对程序中可能发生的用户异常进行捕捉、处理，则程序编译不能通过。

Java 语言将必须被捕捉、处理的用户异常称为**勾选**(checked)异常或受检异常；而将系统异常、编程异常这两种没有被强制要求处理的异常称为**非勾选**(unchecked)异常或非受检异常。

2. 勾选异常的处理

如果一个方法在执行过程中可能抛出**勾选异常**(即因用户因素导致的用户异常)，包括其调用下级方法过程中抛出的勾选异常，则该方法必须对勾选异常进行**捕捉或声明**(catch or specify)。

1）捕捉

捕捉就是在方法体中编写 **try-catch** 语句，捕获并处理本方法或其调用的下级方法所抛出的勾选异常。

假设图 5-14 中的方法 fun1()可能抛出一个 A 类的勾选异常对象 eA，其调用的下级方法 fun2()还可能抛出一个 B 类的勾选异常对象 eB。如果由方法 fun1()负责捕获并处理勾选异常，则程序员应当按如下代码框架来定义方法 fun1()。

```
… fun1() {                        //方法 fun1()负责捕获并处理勾选异常时的代码框架
   …
   try {                          //启用异常处理机制
      …
      if (发现勾选异常 A)   throw eA;//报告异常:抛出一个 A 类的勾选异常对象 eA
      fun2();      //调用方法 fun2(),调用过程中还可能会抛出一个 B 类的勾选异常对象 eB
```

```
        …
    }
    catch ( A  e) {  异常处理代码 1 }      //捕获并处理 A 类的勾选异常
    catch ( B  e) {  异常处理代码 2 }      //捕获并处理 B 类的勾选异常
    …
}
```

2）声明

方法可以自己捕获并处理所抛出的勾选异常，或者将勾选异常交由上级方法处理。如果方法自己不处理勾选异常，而是将它们提交给上级方法处理，则必须向上级方法**声明**这些勾选异常。这就是 Java 语言对勾选异常所制定的捕捉或声明原则。

声明勾选异常，就是在方法头的后面使用关键字 **throws** 给出**勾选异常列表**，向上级方法列出自己可能会抛出哪些勾选异常。如果采用这种声明方式，则程序员应当按如下代码框架来修改 1)中 fun1()的定义代码。

```
… fun1() throws   A, B {  //方法 fun1()通过声明将勾选异常交由上级方法处理时的代码框架
    …
    if (发现勾选异常 A) throw eA;              //报告异常:抛出一个 A 类的勾选异常对象 eA
    fun2();               // 调用方法 fun2(),调用过程中还可能会抛出一个 B 类的勾选异常对象 eB
    …
}
```

方法 fun1()所声明的勾选异常 A、B 仍需要被调用它的上级方法(例如图 5-14 中的主方法 main())捕获并处理，否则上级方法的编译就不能通过。

在图 5-14 所示的方法多级嵌套调用关系中，如果其中的某个方法可能抛出勾选异常，则该方法的每个上级方法都需遵循捕捉或声明原则，直至该勾选异常被捕获并处理。Java 虚拟机会按照方法调用的返回顺序逐级向上，查找能够捕获勾选异常的 catch 子句。如果一直到主方法 main()也没找到能够捕获勾选异常的 catch 子句，则程序的编译不能通过。

简单地说，程序员在调用某个声明了勾选异常的方法时，必须对其所声明的勾选异常进行捕捉或继续声明，否则属于语法错误。Java API 中的某些方法就会抛出勾选异常，程序员在阅读 Java API 说明文档时需要关注方法的异常声明。如果方法声明了勾选异常，则调用时必须进行捕捉或继续声明。

5.6.5 自定义异常类

程序员可以定义自己的异常类，这样就能描述自己程序中可能发生的特定异常情况。例如，身份证号由 18 数字组成(最后一位可能是字母)，假设用户输入了错误的身份证号，程序员可以定义一个 ID 异常类来描述这种异常。例 5-12 给出了一个描述身份证号异常的 ID 异常类示例代码。

例 5-12 一个描述身份证号异常的 ID 异常类示例代码(MyIDException.java)

```
1  class MyIDException extends Exception {  //ID 异常类:继承 Java API 中的异常类 Exception
2      private String ID = null;            //添加字段:存储错误的身份证号
3      public MyIDException(String msg, String id) {  //构造方法
4          super( msg );                    //调用超类 Exception 的构造方法
5          ID = id;
6      }
```

```
7       public String toString() {          //重写 toString()方法,增加 ID 信息
8           return( ID + ":" + super.toString() );
9   } }
```

通常,程序员定义异常类时会选择从 Java API 中的异常类 Exception 或运行时异常类 RuntimeException 继承,然后在此基础上扩展。请注意这两者的区别。

(1) 从异常类 **Exception** 继承。所定义出的子类属于勾选异常,必须遵循捕捉或声明原则进行处理。

(2) 从运行时异常类 **RuntimeException** 继承。所定义出的子类是非勾选异常。针对非勾选异常,Java 语言不做强制要求,程序员可以处理,也可以不做任何处理。

本节习题

1. Java 程序中的语法错误主要通过(　　)来进行排查。
   A. Java 编译器　　B. 运行测试
   C. Java 虚拟机　　D. Java 异常处理机制
2. Java 程序中的语义(逻辑)错误主要通过(　　)来进行排查。
   A. Java 编译器　　B. 运行测试
   C. Java 虚拟机　　D. Java 异常处理机制
3. Java 程序中的运行时错误主要通过(　　)来进行排查。
   A. Java 编译器　　B. 运行测试
   C. Java 虚拟机　　D. Java 异常处理机制
4. 下列选项中,(　　)不属于 Java 异常处理机制的范畴。
   A. 发现异常　　B. 报告异常
   C. 处理异常　　D. 异常对象的垃圾回收
5. 下面的异常类中,(　　)属于必须被捕捉或声明的勾选异常。
   A. Error 类及其子类　　B. RuntimeException 类及其子类
   C. IOException 类及其子类　　D. NullPointerException 类
6. 下列抛出异常对象的语句中,错误的是(　　)。
   A. Exception e = new Exception(); throw e;
   B. throw new Exception();
   C. throw new IOException ();
   D. throw new String();
7. 在 try-catch 语句中,不能被省略的子句是(　　)。
   A. try 子句　　B. catch 子句
   C. finally 子句　　D. 以上 3 个子句都能省略
8. 在 try-catch 语句中,有可能不执行的子句是(　　)。
   A. try 子句　　B. catch 子句
   C. finally 子句　　D. 以上 3 个子句都有可能

5.7 泛型与数据集合类

泛型(generics)是从 JDK 1.5 开始引入的一种新特性,其目的是通过**类型参数化**来提高程序代码的重用性。类型参数化就是将类、接口或方法所处理数据的类型抽象成参数,这样可以定义出**泛型类**、**泛型接口**或**泛型方法**。同一个泛型类、泛型接口或泛型方法可以处理多种不同类型的数据,称其**代码**可以被不同数据类型**重用**。

5.7.1 类型参数化

本节通过一个具体的程序实例讲解什么是类型参数化,以及如何利用类型参数化来提高程序代码的重用性。例 5-13 给出了两个分别存放 Integer 型数据和 Double 型数据的集合类示例代码。

例 5-13 两个分别存放 Integer 型数据和 Double 型数据的集合类示例代码

```
1  class IntegerSet {                          //Integer 型集合类
2        public Integer set[];                 //用于存放 Integer 型数据的数组
3        public IntegerSet( Integer p[])       //构造方法
4        {   set = p;   }
5        public void show() {                  //显示数据集合中的元素
6            for (int n = 0; n < set.length; n++)
7                  System.out.print( set[n]  + " " );
8            System.out.println();
9  }  }
```

```
1  class DoubleSet {                           //Double 型集合类
2        public Double set[];                  //用于存放 Double 型数据的数组
3        public DoubleSet( Double p[])         //构造方法
4        {   set = p;   }
5        public void show() {                  //显示数据集合中的元素
6            for (int n = 0; n < set.length; n++)
7                  System.out.print( set[n]  + " " );
8            System.out.println();
9  }  }
```

可以使用例 5-13 中的两个类分别定义出 Integer 型集合对象和 Double 型集合对象。例如:

```
//定义一个 Integer 型集合对象
Integer ia[] = { 10, 20, 30 };
IntegerSet is = new IntegerSet( ia );
is.show();             //显示集合对象 is 中的元素,显示结果:10 20 30
//定义一个 Double 型集合对象
Double da[] = { 10.5, 20.5, 30.5 };
DoubleSet ds = new DoubleSet( da );
ds.show();             //显示集合对象 ds 中的元素,显示结果:10.5 20.5 30.5
```

仔细分析例 5-13 中的 Integer 型集合类和 Double 型集合类,可以看出这两个类的功能

完全相同,唯一不同的是集合元素的数据类型不一样,一个是 Integer 型,另一个是 Double 型。将这两个类合并成一个类,就可以有效降低程序的编码工作量。

Java 语言可以将 Integer、Double 等具体数据类型抽象成一个类型参数(称为**类型形参**),这就是**类型参数化**。假设将类型形参命名为 T,利用类型形参 T 可以将 Integer 型集合类和 Double 型集合类合并成一个 T 类型的集合类,这就是一个**泛型类**。这里的类型形参 T 可以指代任意一种具体的数据类型,或者说类型形参 T 是一种通用数据类型(称为**泛型**)。例 5-14 给出一个 T 类型的泛型集合类示例代码。

例 5-14 一个 T 类型的泛型集合类示例代码

```
1  class GenericSet < T > {          //泛型集合类:类型形参 T 可以指代任意一种具体的数据类型
2        public T set[];             //用于存放 T 类型数据的数组
3        public GenericSet( T p[]) //构造方法
4        {  set = p;  }
5        public void show() {       //显示数据集合中的元素
6            for (int n = 0; n < set.length; n++)
7                System.out.print( set[n] + " " );
8            System.out.println();
9  }  }
```

使用例 5-14 中的泛型集合类 GenericSet < **T** >时,需要明确给出类型形参 T 所指代的具体数据类型(称为**类型实参**)。不同类型实参表示不同类型的集合类。例如:

- GenericSet < **Integer** >表示 Integer 类型的集合类,类型实参为 Integer。
- GenericSet < **Double** >表示 Double 类型的集合类,类型实参为 Double。

可以使用这两个类分别定义出 Integer 型集合对象或 Double 型集合对象。例如:

```
//定义一个 Integer 型集合对象
Integer ia[] = { 10, 20, 30 };
GenericSet < Integer > is = newGenericSet < Integer >( ia );  //类型实参为 Integer
//或:GenericSet < Integer > is = new GenericSet <>( ia ); //可省略第 2 个类型实参 Integer
is.show();                                        //显示集合对象 is 中的元素:10 20 30
//定义一个 Double 型集合对象
Double da[] = { 10.5, 20.5, 30.5 };
GenericSet < Double > ds = new GenericSet < Double >( da );    //类型实参为 Double
ds.show();                                   //显示集合对象 ds 中的元素:10.5 20.5 30.5
```

还可以使用例 5-14 中的泛型类 GenericSet < **T** >定义出其他类型的集合对象。例如:

```
//定义一个 Short 型集合对象
Short sa[] = { 10, 20, 30 };
GenericSet < Short > ss = new GenericSet < Short >( sa );     //类型实参为 Short
ss.show();                                        //显示集合对象 ss 中的元素:10 20 30
//定义一个 Float 型集合对象
Float fa[] = { 10.5f, 20.5f, 30.5f };
GenericSet < Float > fs = new GenericSet < Float >( fa );     //类型实参为 Float 型
fs.show();                                   //显示集合对象 fs 中的元素:10.5 20.5 30.5
```

使用泛型类 GenericSet < **T** >可以定义出各种不同数据类型的集合类。换句话说,泛型

类 GenericSet < **T** >是一种能够被重用的代码，它可以被不同的数据类型重用。利用类型参数化，可以有效提高程序代码的重用性。

5.7.2 泛型编程

通过 5.7.1 节的集合类程序例子，读者已经直观地了解了什么是类型参数化，什么是泛型类以及泛型类的使用方法。使用泛型类可以定义出各种不同数据类型的具体类。

Java 语言中，带类型参数的类称为**泛型类**。同理，带类型参数的接口称为**泛型接口**，带类型参数的方法称为**泛型方法**。在了解了泛型的基本概念之后，读者就可以使用 Java API 中的泛型类、泛型接口或泛型方法来编写程序了。例如，读者可以使用 Java API 提供的数据集合类来编写复杂的数据集合处理程序。

5.7.1 节讲解的是如何**使用**别人编写的泛型类。本节所要讲解的是如何**编写**自己的泛型类，即**泛型编程**。**注**：泛型编程是一种高级 Java 编程技术，内容比较难。初学者可跳过本节，这不会影响后续内容的学习。

1. 泛型类或泛型接口

这里给出定义泛型类或泛型接口的 Java 语法，读者可对照 5.7.1 节所讨论的泛型集合类 GenericSet < T >来理解泛型语法的应用语境。

Java 语法：定义**泛型类**或**泛型接口**

```
class 泛型类名<类型形参列表>  {   类成员   }
interface 泛型接口名<类型形参列表>  {   接口成员   }
```

语法说明：

- 定义泛型类、泛型接口时，需在类名或接口名后面用一对尖括号“<>”给出**类型形参列表**。
- **类型形参**是一种表示数据类型的参数。多个类型形参之间用逗号“，”隔开，例如 < T >、< T1，T2 >、< K，V >等。类型参数名需符合标识符的命名规则，习惯上用 T、E、K、N、V 等表示。
- 泛型类（或泛型接口）定义代码的其余部分与普通类（或普通接口）一样。所不同的是，类型形参就像是一种新的数据类型，可以用来定义字段成员或方法成员中的形参、局部变量或返回值类型。
- 使用泛型类（或泛型接口）时，需明确给出类型形参所指代的**类型实参**（即某一种具体的数据类型）。指定了类型实参的泛型类（或泛型接口）称为**具体类**（或**具体接口**）。**请注意**，类型实参只能是**引用数据类型**（例如类、接口、数组等），不能是基本数据类型（例如 int、double 等）。
- 如果对类型形参不做限定（例如< **T** >），则类型形参可以指代任意一种引用数据类型，即使用泛型类（或泛型接口）时的类型实参可以是任意一种引用数据类型。如果希望将类型实参限定在某个**类族**或**接口族**范围内，则需要按如下格式来定义类型形参。

```
T extends 超类名       //类型形参 T 可以指代某个超类及其所有子类
T extends 接口名       //类型形参 T 可以指代任意实现了该接口的类,或从其扩展出的子接口
```

- 使用泛型类(或泛型接口)可以定义出不同数据类型的具体类(或具体接口)。换句话说,泛型类(或泛型接口)是一种能被重用的代码,它可以被不同的数据类型重用。

使用例 5-14 给出的泛型集合类 GenericSet < T >可以定义出不同的具体类,例如 GenericSet < **Integer** >表示 Integer 型的集合类,GenericSet < Double >表示 Double 型的集合类,GenericSet < Object >表示 Object 型的集合类,等等。

泛型集合类 GenericSet < T >没有对类型形参 T 做任何限制,因此使用该类的类型实参可以是任意一种引用数据类型。例如,下列定义集合类对象的语句都是正确的。

```
GenericSet < Integer > is = new GenericSet < Integer >( … ); //定义一个 Integer 型集合对象
GenericSet < Double > ds = new GenericSet < Double >( … );   //定义一个 Double 型集合对象
GenericSet < Character > cs = new GenericSet < Character >( … ); //定义一个 Character 型集合
                                                              //对象
GenericSet < String > ss = new GenericSet < String >( … );   //定义一个 String 型集合对象
GenericSet < Object > os = new GenericSet < Object >( … );   //定义一个 Object 型集合对象
```

注:上述语句小括号中的"…"表示被省略的集合初始值。

如果希望将类型实参限定在某个**类族**或**接口族**范围内,例如限定在数值类 Number 及其子类范围内,则例 5-14 中的泛型集合类需要按如下格式来定义类型形参 T。

```
class GenericSet < T extends Number > {        //类型形参 T 只能指代数值类 Number 或其子类
    …                                          //其余代码不变,省略
}
```

使用这个泛型集合类 GenericSet < T extends Number >只能定义数值型的集合对象。例如,下列定义集合对象语句中的类型实参必须是数值类 Number 或其子类。

```
GenericSet < Number > ns = new GenericSet < Number >( … );   //正确:定义一个 Number 型集合对象
GenericSet < Integer > is = new GenericSet < Integer >( … ); //正确:Integer 是 Number 的子类
GenericSet < Double > ds = new GenericSet < Double >( … );   //正确:Double 是 Number 的子类
GenericSet < Character > cs = new GenericSet < Character >( … );
                                                            //错误:Character 不是 Number 的子类
GenericSet < String > ss = new GenericSet < String >( … );   //错误:String 不是 Number 的子类
GenericSet < Object > os = new GenericSet < Object >( … );   //错误:Object 不是 Number 的子类
```

2. 泛型族

这里以 Java API 中的数值类 Number 为例,具体讲解什么是泛型族。数值类 Number 有 6 个子类,分别是 Byte、Short、Integer、Long、Float 和 Double,它们构成一个以类 Number 为根类的数值类族。

1) 泛型族介绍

基于**同一泛型类**为某个**类族或接口族**中的每个类分别定义出一个具体的类,这些具体类合在一起称为一个**泛型族**。例 5-15 给出一个简单的泛型类 A < T >,下面讨论如何基于这个泛型类 A < T >定义出一个泛型族。

例 5-15　一个简单的泛型类 A<T>示例代码

```
1  class A<T> {                          //一个简单的泛型类 A
2          public  T a;                  //字段:T 类型
3          public A(T x) {  a = x;  }    //构造方法
4  }
```

可以基于泛型类 A<T>为数值类族中的根类 Number 及其 6 个子类分别定义一个具体类，即 A < **Number** >、A < **Byte** >、A < **Short** >、A < **Integer** >、A < **Long** >、A < **Float** >、A < **Double** >，这 7 个具体类就组成了一个基于泛型类 A<T>的数值类泛型族。

2）通配符类型的引用变量

可以使用泛型族中的具体类定义引用变量或创建对象。例如：

```
A< Integer > ia ;            //定义 A< Integer >类的引用变量 ia
ia = new A< Integer >(10); //创建一个 A< Integer >类的对象,将其引用赋值给引用变量 ia
```

这里，A<Integer>类的引用变量 ia 引用的是一个同类型对象，即 A<Integer>类的对象。

Java 语言中，超类或接口的引用变量可以引用其子类的对象，其目的是为了让类族或接口族中的类共用算法代码（对象替换与多态机制）。Java 语言也专门为泛型族设计了 3 种用问号“?”表示的**通配符类型**（wildcard type）引用变量，其目的是为了让**泛型族**共用算法代码。

（1）泛型名<?>：可引用基于泛型类或泛型接口所定义出的任何具体类的对象。例如：

```
A<?>  ref ;
```

该语句定义了一个 A <?>通配符类型的引用变量 ref，它可以引用 A < Integer >、A < Double >、A < String >、A < Object >等任何具体类的对象。

（2）泛型名<? extends 类名>：只能引用基于泛型类或泛型接口由某个类及其子类所定义出的具体类的对象，即只能引用某个泛型族中类的对象。例如：

```
A< ? extends Number >  ref ;
```

该语句定义了一个 A <? extends **Number** >通配符类型的引用变量 ref，它只能引用 A < Number >、A < Integer >、A < Double >等由类 Number 及其子类所定义出的数值类泛型族中的具体类对象。

（3）泛型名<? super 类名>：只能引用基于泛型类或泛型接口由**某个**类及其超类所定义出的具体类的对象。例如：

```
A< ? super Integer >  ref ;
```

该语句定义了一个 A <? super **Integer** >通配符类型的引用变量 ref，它只能引用 A < Integer >、A < Number >、A < Object >等由类 Integer 及其超类所定义出的具体类的对象。

通配符类型的语法细则：通配符类型可用于定义方法的**形参**或**返回值类型**，或定义方法中的**局部引用变量**，也可用于定义类中的**字段成员**，但不能用于创建对象。

3) 泛型族共用算法代码

通过对象替换与多态机制,同一类族或接口族中的类可以共用算法代码。Java 语言还通过对象替换与多态机制,再结合通配符类型,可以继续让同一泛型族中的类共用算法代码。

方法是描述某种数据处理算法的代码,方法中形参的数据类型决定了算法能够处理哪种类型的数据。例 5-15 曾给出一个简单的泛型类 A < T >,假设有如下一个显示方法 show():

```
void show( A < Integer > aRef )             //形参 aRef 为 A < Integer >类的对象引用
{  System.out.println( aRef.a );  }         //显示 A < Integer >类对象的字段成员 a
```

显示方法 show()只能处理 A < Integer >类的对象,例如:

```
show( new A < Integer >(5) );               //处理 A < Integer >类的对象,显示结果:5
```

如果将显示方法 show()的形参类型改为通配符类型,则可以让这个方法被某个泛型族共用。例如,按如下形式修改方法 show()中形参 aRef 的数据类型:

```
void show( A < ? extends Number >  aRef ) //aRef 可引用基于 A < T >的 Number 泛型族中的所有对象
{  System.out.println( aRef.a );  }
```

修改后的方法 show()可以处理基于泛型类 A < T >的数值类 Number 泛型族中的所有对象,即数值类泛型族可以共用方法 show()的算法代码。例如:

```
show( new A < Integer >(5) );               //处理 A < Integer >类的对象,显示结果:5
show( new A < Double >(5.5) );              //处理 A < Double >类的对象,显示结果:5.5
show( new A < Float >(5.5f) );              //处理 A < Float >类的对象,显示结果:5.5
```

3. 泛型类的继承与扩展

泛型类可以被继承、扩展,扩展时可以继续增加类型形参。例 5-16 定义了两个泛型类 B1 < T >和 B2 < T1, T2 >。这两个类继承并扩展了例 5-15 中的泛型类 A < T >,它们是泛型类 A < T >的**泛型子类**。反过来,泛型类 A < T >被称为是泛型类 B1 < T >、B2 < T1, T2 >的**泛型超类**。

例 5-16 继承泛型类 A < T >所扩展出的两个泛型子类 B1 < T >、B2 < T1, T2 >示例代码

```
1  class B1 < T > extends A < T > {        //定义泛型类 B1 < T >时继承泛型类 A < T >
2         public  T b;                      //新增成员
3         public B1(T x, T y) {             //构造方法
4               super(x);                   //调用超类的构造方法
5               b = y;
6  }  }
```

```
1  class B2 < T1, T2 > extends A < T1 > { //定义泛型类 B2 < T1, T2 >时继承泛型类 A < T >
2         public  T2 b;                     //新增成员
3         public B2(T1 x, T2 y) {           //构造方法
4               super(x);                   //调用超类的构造方法
5               b = y;
6  }  }
```

1）泛型类 B1 < T >

例 5-16 中，泛型类 B1 < T >是泛型类 A < T >的子类。基于泛型类 A < T >可以定义出不同类型的具体类，例如 A < Number >、A < Integer >、A < Double >等，它们构成了一个基于泛型超类 A < T >的泛型族（参见图 5-15）。

同样，基于泛型类 B1 < T >也可以定义出不同类型的具体类，例如 B1 < Number >、B1 < Integer >、B1 < Double >等，它们构成了一个基于泛型子类 B1 < T >的泛型族。这两个泛型族中的具体类之间存在什么样的继承关系呢？

- 同一泛型族中的具体类之间不存在继承关系。例如，虽然 Integer 是 Number 的子类，但 A < Integer >不是 A < Number >的子类。同理，B1 < Integer >也不是 B1 < Number >的子类。
- 泛型超类与泛型子类的同类型具体类之间存在继承关系。例如 B1 < Number >是 A < Number >的子类、B1 < Integer >是 A < Integer >的子类等。

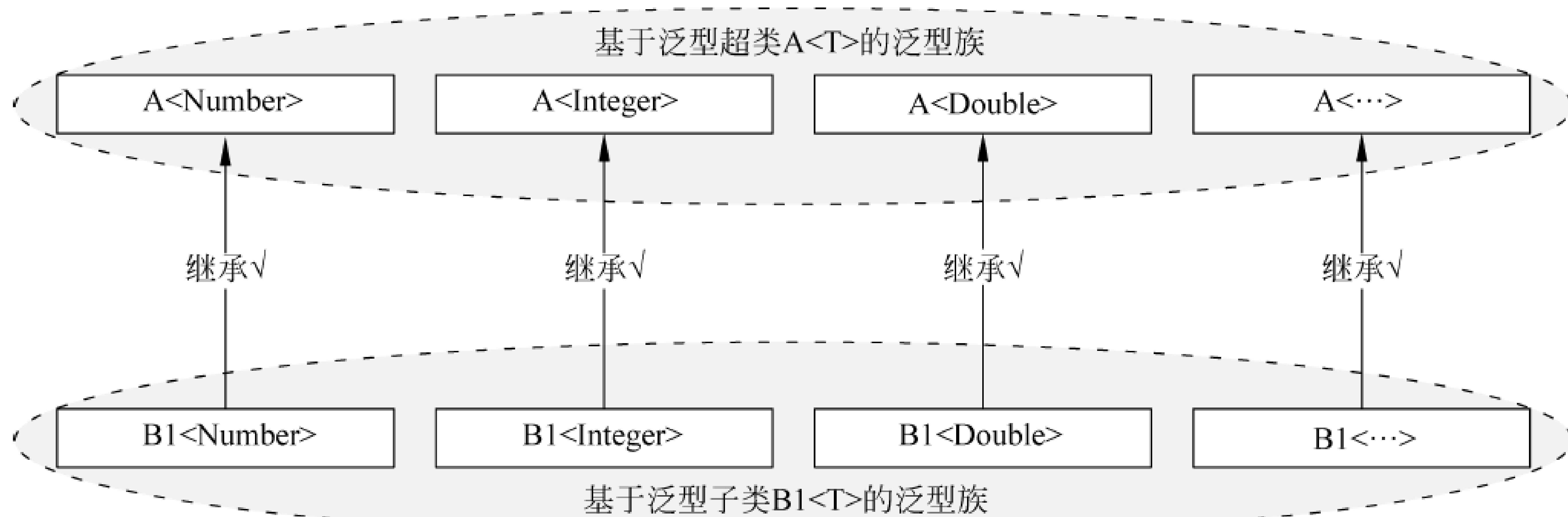


图 5-15　基于泛型超类 A < T >的泛型族与基于泛型子类 B1 < T >的泛型族

2）泛型类 B2 < T1，T2 >

例 5-16 中，泛型类 B2 < T1，T2 >也是泛型类 A < T >的子类。但与 B1 < T >不同的是，泛型类 B2 < T1，T2 >有两个类型参数。对基于泛型超类 A < T >的泛型族和基于泛型子类 B2 < T1，T2 >的泛型族，图 5-16 给出了这两个泛型族中具体类之间的继承关系图。

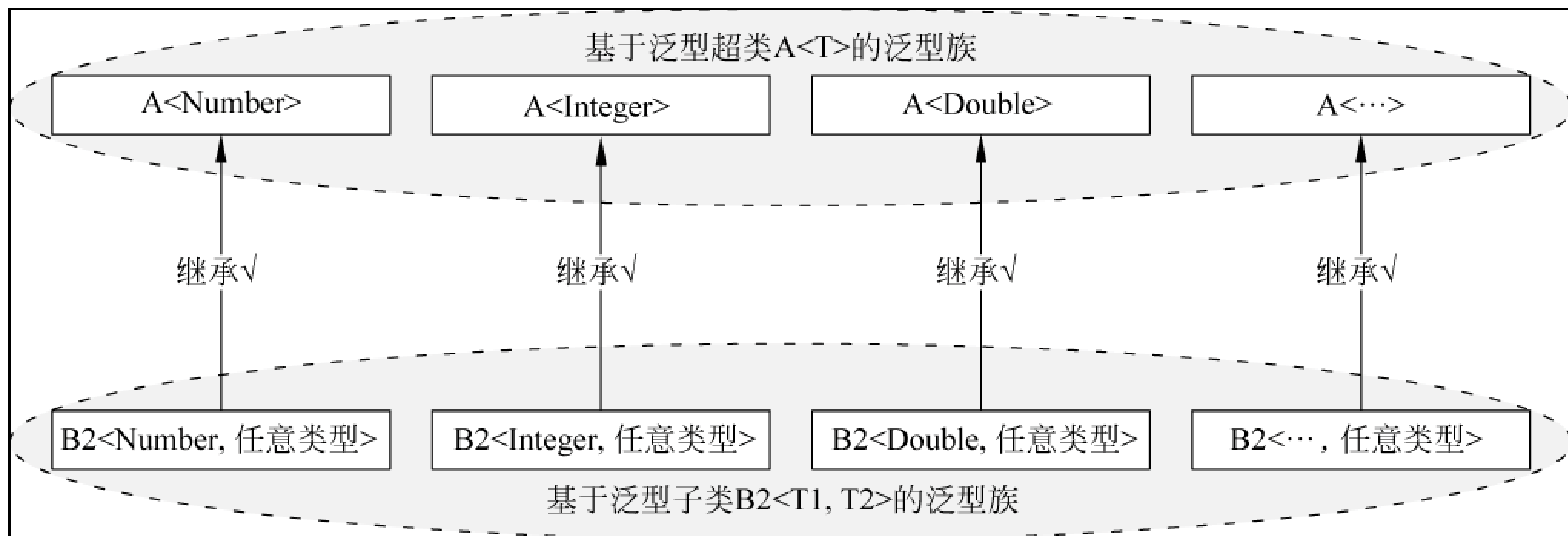


图 5-16　基于泛型超类 A < T >的泛型族与基于泛型子类 B2 < T1，T2 >的泛型族

4. 泛型方法

可以将类中的某个方法成员单独定义成一个**泛型方法**。例如:

```
class A {                                    //将类 A 中的方法成员 show()单独定义成一个泛型方法
      …                                      //其他代码省略
      public static < T > void show(T obj)   //泛型方法:显示 T 类型的对象
      {   System.out.println( obj.toString() );   }
}
```

其中,类型形参 T 可指代任意一种具体的引用数据类型。方法 show()可理解为是一个能够处理任意引用类型数据的泛型方法。定义泛型方法时,需在紧邻返回值的前面指定类型形参。可以为泛型方法指定多个类型形参,多个类型形参之间用逗号“,”隔开。

调用泛型方法时,Java 编译器会自动根据所处理对象的类型推断出类型形参所指代的具体数据类型(即类型实参),程序员不需要显式指定。例如:

```
A.show( new Integer(5) );      //编译器可推断出类型形参 T 指代的是 Integer 类型,显示结果:5
A.show( new Double(5.5) );     //编译器可推断出类型形参 T 指代的是 Double 类型,显示结果:5.5
```

5.7.3 数据集合

数据集合就是一组数据的集合。例如,表 5-4 所示的学生成绩单就是一组关于学生姓名和成绩的数据集合。

表 5-4 学生成绩单

姓　　名	成　　绩
张三	92
李四	86
王五	95
…	…

关于数据集合,存在如下 3 个层次。

(1) **数据项**(data item)。数据项是数据集合中的最小单位。例如表 5-4 中第 2 行的姓名“张三”、成绩“92”都分别是一个的数据项。数据项也称作**字段**(field)。

(2) **数据元素**(data element)。数据元素是由多个具有内在关联关系的数据项组成。例如,表 5-4 中第 2 行“张三,92”表示张三的成绩是 92,“张三”和“92”这两个数据项具有内在关联关系,它们就构成了一个数据元素。一个数据元素也称作一条**记录**(record)。

(3) **数据集合**(data set)。数据集合由多个并列的数据元素所组成。例如,表 5-4 就是一个由多个数据元素(每一行就是一个数据元素)组成的数据集合。一个数据集合也称作一张**表**(table)。

如果数据集合中各数据元素之间的逻辑关系是有序的,则称该数据集合为有序数据集合,否则称为无序数据集合。对数据集合的处理通常包括增加、查找、修改或删除数据元素,这被统称为**增查改删**(Create、Read、Update、Delete,CRUD)。为了提高查找速度,可以将数据集合中的数据元素按照某种规则事先进行排序。

编写一个处理数据集合的计算机程序,要涉及两个方面的内容:一是如何组织和存储数据集合,即数据结构;二是如何基于数据结构进行数据处理,即算法。同样的数据集合,存储的数据结构不同,将导致处理的算法也会有所不同。不同数据结构适用于不同的算法。计算机学科中的**数据结构**(data structure)就是专门研究数据结构与算法关系的课程。数据结构的研究对象就是数据集合,其研究内容是如何对数据集合进行组织、存储和处理的一般方法。

例 5-17 给出一个存储和处理表 5-4 中学生成绩单的 Java 示例代码。其中的学生类 Student 实现一个 Comparable 接口,通过 compareTo()方法定义了比较两个学生对象大小的规则。学生类 Student 还重写了 hashCode()方法和 equals()方法,用于比较两个学生对象的内容是否相等。

例 5-17　一个存储和处理学生成绩单的 Java 示例代码(StudentScoreTest.java)

```
1   public class StudentScoreTest {                    //主类
2       public static void main(String[] args) {       //主方法
3           Student sa[] = new Student[3];             //创建一个保存 3 名学生成绩的对象数组
4           sa[0] = new Student( "张三", 92 );         //添加数据元素
5           sa[1] = new Student( "李四", 86 );
6           sa[2] = new Student( "王五", 95 );
7           for (int n = 0; n < sa.length; n++)       //遍历数组,显示学生成绩单
8               System.out.println( sa[n].toString() );
9           //比较两个对象的大小:调用对象的 compareTo()方法
10          System.out.println( sa[0].compareTo(sa[1]) );  //比较 sa[0]和 sa[1]的大小,显示 6
11          System.out.println( sa[0].compareTo(sa[2]) );  //比较 sa[0]和 sa[2]的大小,显示 -3
12          //比较对象的内容是否相等:hashCode()可用于快速比较,equals()则是全面比较
13          Student s = new Student("赵六", 92);   //再创建一个与 s[0]成绩相同的对象 s
14          System.out.println( sa[0].hashCode() == s.hashCode() );   //显示结果为 true
15          System.out.println( sa[0].equals(s) );      // 同时比较成绩和姓名,显示结果为 false
16  } }
17
18  class Student implements Comparable<Student> {  //学生成绩类
19      private String name;                          //姓名
20      private int score;                            //成绩
21      public Student(String p1, int p2)             //构造方法
22      {  name = p1;  score = p2;  }
23      public String toString()                      //重写 toString()方法
24      {  return String.format("%s: %d", name, score);  }
25      public int compareTo(Student s) {        //实现 Comparable 接口所规定的比较大小方法
26          //假设比较两个学生大小的规则是先比较成绩,成绩相同时再比较姓名
27          int n = score - s.score;                  //先比较成绩
28          if (n != 0)  return n;                    //成绩不同时,直接返回成绩比较的结果
29          else  return name.compareTo(s.name);  //成绩相同时,再比较姓名
30      }
31      public int hashCode() {                       //重写 hashCode()方法
32      {  return score;  }                       //生成哈希码:简单地将成绩作为学生的哈希码
33      public boolean equals(Object obj) {          //重写 equals()方法
```

```
34        if ((obj instanceof Student) == false)  return false;  //类型不同,则直接返回 false
35        Student s = (Student)obj;
36        if(score != s.score)  return false;          //先判断成绩,成绩不同则学生不同
37        if(name.equals(s.name) != true)  return false;  //再判断姓名是否相同
38        return true;               //姓名和成绩都相同时,两个对象的内容就相同,返回 true
39  }  }
```

例 5-17 使用对象数组来存储学生成绩的数据集合。实际应用中,对学生成绩这个数据集合还会有很多后续处理,例如增查改删,或排序等,程序员还需要为这些处理编写大量的算法代码。

为方便程序员,Java API 提供了很多具有不同功能的数据集合类。使用这些类,程序员能够快速编写出存储和处理数据集合的 Java 程序。下一节将具体介绍 Java API 中的这些数据集合类。

5.7.4 Java API 中的数据集合类

使用 Java API 所提供的数据集合类可以实现**动态数组**(或称为可变长数组)、**队列**或**堆栈**、**集合**、**映射**(或称为字典)等数据存储功能。另外 Java API 还配套提供了相关的**增查改删**和**排序**算法。

Java API 对数据集合类进行了再次抽象,提炼出 **Collection**(集合)和 **Map**(映射)两个接口,然后按功能要求由数据集合类具体实现这两个接口。Java API 这么做的目的是让多个不同的数据集合类实现相同的接口,这样就构成了一个接口族。同一接口族中的类是可以共用算法代码的。

Java API 在定义集合接口 Collection 和映射接口 Map 时还运用了泛型编程技术,这样可以基于同一泛型接口定义出不同数据类型的具体集合类,例如 Integer 型集合、Double 型集合、String 型集合、Student 型集合等,参见 5.7.1 节的例 5-14。这些不同数据类型的具体集合类构成一个泛型族,同一泛型族中的类也是可以共用算法代码的。

请读者阅读下面的集合接口 Collection<E>和映射接口 Map<K, V>说明文档,了解其中为数据集合操作所定义的一些常用方法接口。

java.util.**Collection**<**E**>接口说明文档

public interface **Collection**<**E**>
extends **Iterable**<**E**>

	修 饰 符	接口成员(节选)	功 能 说 明
1		boolean **add**(E e)	添加一个元素
2		boolean **contains**(Object o)	是否包含某个指定的元素
3		boolean **hasNext**()	是否还有下一个元素
4		E **next**()	返回下一个元素并后移一个元素
5		void **remove**()	删除迭代器当前指向的元素
6		int **size**()	返回元素的个数

续表

	修　饰　符	接口成员(节选)	功 能 说 明
7		boolean **isEmpty**()	集合是否为空
8		void **clear**()	删除所有的元素
9		Object[] **toArray**()	返回一个数组形式的集合
10	default	Stream < E > **stream**()	返回一个流形式的集合
11		Iterator < E > **iterator**()	返回一个迭代器
		…	

java.util.**Map** < **K,V** >接口说明文档			
public interface **Map** < **K,V** >			
	修　饰　符	接口成员(节选)	功 能 说 明
1	static	interface Map.**Entry** < K,V >	内部接口,表示一个键值对
2		V **put**(K key, V value)	添加一个键值对
3		V **get**(Object key)	读取某个键的值
4	default	V **replace**(K key, V value)	修改某个键的值
5	default	V **remove**(Object key)	删除某个键值对
6		boolean **containsKey**(Object key)	是否包含指定的键
7		boolean **containsValue**(Object value)	是否包含指定的值
8		int **size**()	返回键值对的个数
9		boolean **isEmpty**()	映射是否为空
10		void **clear**()	删除所有的键值对
		…	

下面具体讲解 Java API 中常用的数据集合类。

1. 数组列表类 ArrayList < E >

Java API 中的数组列表类 **ArrayList** < E >实现了集合接口 Collection < E >,可实现动态数组的功能。

1) 创建数组列表

例 5-18 给出一个使用类 ArrayList < E >创建数组列表的 Java 示例代码。

例 5-18　一个使用类 ArrayList < E >创建数组列表的 Java 示例代码(ArrayListTest.java)

```
1  import java.util.ArrayList;                              //导入数组列表类 ArrayList
2  public class ArrayListTest {                             //测试类
3      public static void main(String[] args) {             //主方法
4          ArrayList< Integer > v1 = new ArrayList< Integer >();    //Integer 型数组列表
5          v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );   //添加元素
6          for (int n = 0; n < v1.size(); n++)      //使用序号(下标)遍历显示各元素
7              System.out.print( v1.get(n) + ", " );
8          System.out.println();
9
10         ArrayList< Student > v2 = new ArrayList<>();             //Student 型数组列表
```

```
11            v2.add( new Student("张三", 92) );          //添加元素
12            v2.add( new Student("李四", 86) );
13            v2.add( new Student("王五", 95) );
14            for (int n = 0; n < v2.size(); n++)          //使用序号(下标)遍历显示各元素
15              System.out.println( v2.get(n) );
16  }  }
```

在 Eclipse 集成开发环境中运行例 5-18 的程序,运行结果如图 5-17 所示。

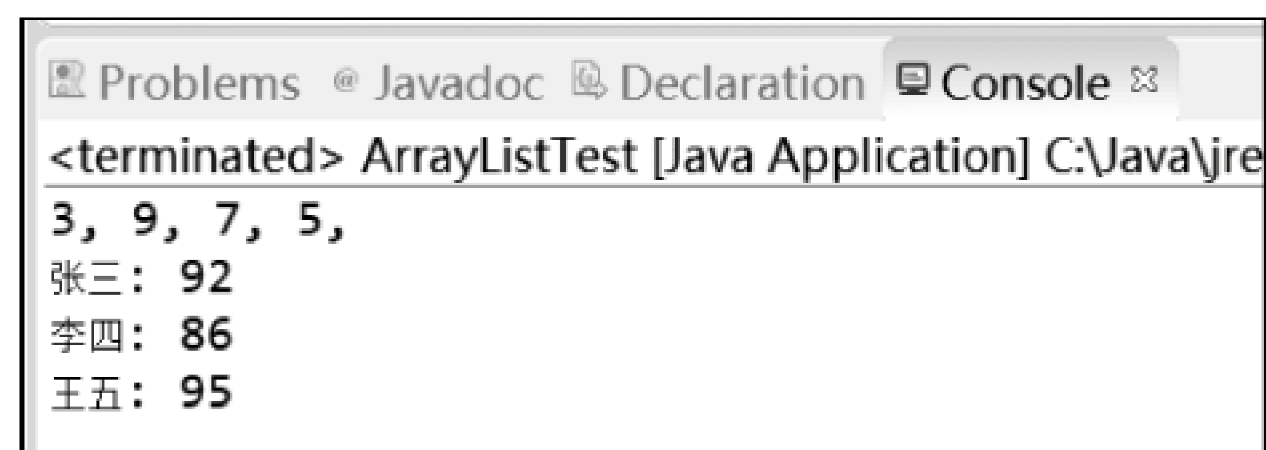


图 5-17　例 5-18 程序的运行结果

2）使用增强 for 语句遍历数组列表

除了常规 for 语句之外,程序员还可以使用增强 for 语句遍历数组列表中的元素。例如:

```
ArrayList< Integer > v1 = new ArrayList< Integer >();     //Integer 型数组列表
v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );   //添加元素
for ( Integer e : v1)                                      //使用增强 for 语句遍历显示各元素
      System.out.print( e + ", " );
```

使用增强 for 语句遍历数组列表 v1 的显示结果为:

```
3, 9, 7, 5,
```

3）使用迭代器遍历数组列表

遍历数组列表的第 3 种方法是**迭代器**(iterator)。迭代器是 Java API 提供的一种遍历数据集合的模式,可调用数据集合类的 **iterator**()方法获得数据集合的迭代器对象。迭代器对象描述了遍历数据集合时的元素位置信息,如图 5-18 所示。

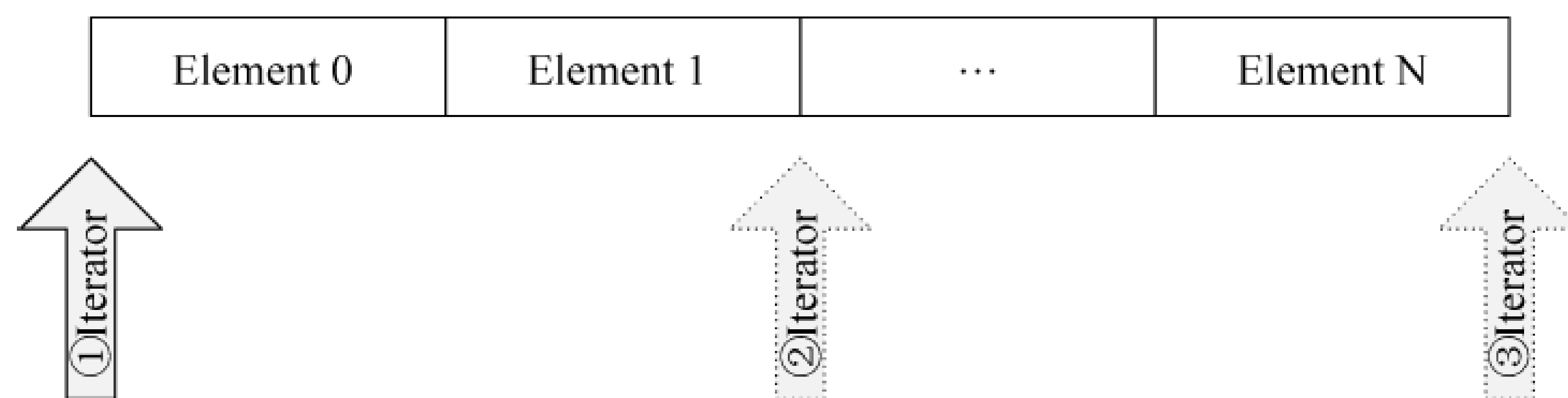


图 5-18　数据集合的迭代器对象

假设有一个如下的数组列表 v1:

```
ArrayList< Integer > v1 = new ArrayList< Integer >();      //Integer 型数组列表
v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );    //添加元素
```

使用迭代器遍历数据集合 v1 的代码框架如下:

```
Iterator < Integer > iv1 = v1.iterator();  //获取 v1 的迭代器对象,此时迭代器处于图 5-18
                                           //中①的位置
while ( iv1.hasNext() ) {          //如果有下一个元素则为 true,例如图 5 - 18 中①、②的位置
    Integer e = iv1.next();        //取出下一个元素,然后迭代器自动后移一个元素
    System.out.print( e + ", " );
}  //当迭代器处于图 5 - 18 中③的位置时,hasNext()返回 false,循环结束
```

Java API 中所有实现 Collection < E >接口的数据集合类都可以使用迭代器进行遍历操作。使用迭代器,不同数据集合类的遍历代码框架都是一样的。换句话说,不同的数据集合类可以通过迭代器共用遍历算法代码。

Java API 中的迭代器 **Iterator** 是一个泛型接口。请读者阅读下面的迭代器接口 Iterator < E >说明文档。

java. util. **Iterator** < E >接口说明文档			
public interface **Iterator** < **E** >			
	修　饰　符	接口成员(全部)	功 能 说 明
1		boolean **hasNext**()	是否还有下一个元素
2		E **next**()	返回下一个元素并后移一个元素位置
3	default	void **remove**()	删除迭代器当前指向的元素
4	default	void **forEachRemaining** (Consumer <? super E > action)	遍历并处理集合中剩余的元素

4) 使用 Collections 类中的静态方法处理数组列表

为了方便程序员,Java API 提供了一个专门的数据集合算法类 **Collections**,其中包括各种数据集合处理算法。这些处理算法被定义成静态成员,可直接通过类名 Collections 进行调用。例 5-19 给出一个使用 Collections 类中静态方法处理数组列表的 Java 示例代码。

例 5-19　一个使用 Collections 类中静态方法处理数组列表的 Java 示例代码(ArrayListTest. java)

```
1   import java.util.ArrayList;                              //导入数组列表类 ArrayList
2   import java.util.Collections;                            //导入集合算法类
3   public class ArrayListTest {                             //测试类
4       public static void main(String[] args) {             //主方法
5           ArrayList < Integer > v1 = new ArrayList<>();    //Integer 型数组列表
6           v1.add( 3 );  v1.add( 9 );  v1.add( 7 );  v1.add( 5 );  //添加元素
7           System.out.println( v1 );                        //直接显示整个数组列表
8           Collections.sort( v1 );                          //对元素进行排序
9           System.out.println( v1 );
10          Collections.reverse( v1 );                       //逆序排列所有元素
11          System.out.println( v1 );
12          System.out.println( Collections.max( v1 ) );    //求数据集合中元素的最大值
13  }  }
```

在 Eclipse 集成开发环境中运行例 5-19 的程序,运行结果如图 5-19 所示。

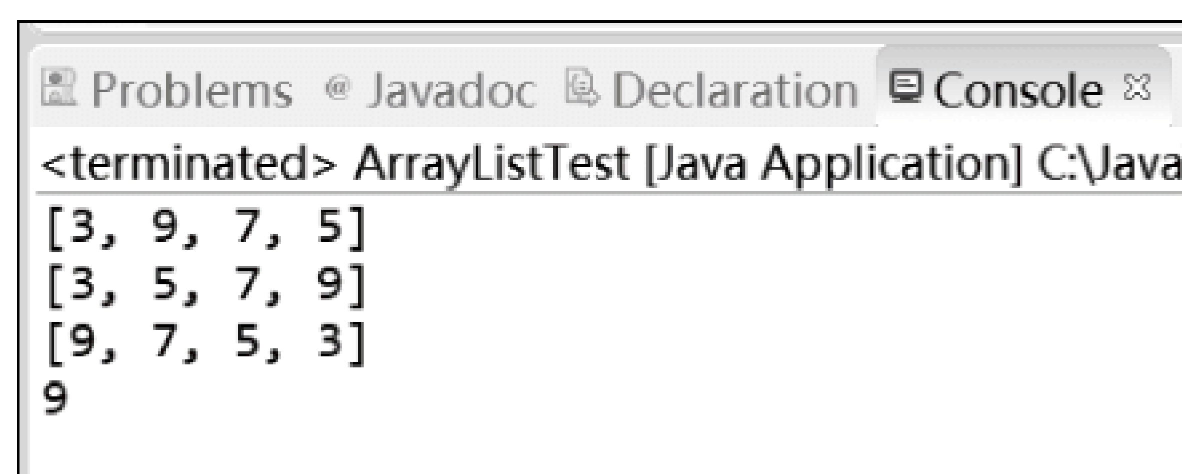


图 5-19　例 5-19 程序的运行结果

2. 双端队列类 LinkedList < E >

Java API 中的双端队列类 **LinkedList** < E >实现了集合接口 Collection < E >。使用这个双端队列类可以实现**队列**的功能,也可以实现**堆栈**的功能。队列和堆栈是两种存储数据集合时常用的数据结构。

如果将数据集合看作是一组数据元素的有序队列,则**队列**(queue)就是在添加元素时总是被加在**队列尾**,取出元素时总是取出排在队列最前面(**队列头**)的那个元素,这种排序规则称为**先进先出**(First-In-First-Out,**FIFO**)。双端队列类 LinkedList < E >中实现队列功能的两个主要方法成员如下。

- **offer**():在队列尾添加一个元素。
- **poll**():取出并删除队列头的元素。

如果将数据集合看作是一组数据元素的有序堆叠,则**堆栈**(stack)就是在添加元素时总是被放在堆栈的顶部(**栈顶**),取出元素时也总是取出栈顶(即最后被放入堆栈)的那个元素,这种排序规则称为**后进先出**(Last-In-First-Out,**LIFO**)。双端队列类 LinkedList < E >中实现堆栈功能的两个主要方法成员如下。

- **push**():向堆栈中压入一个元素。
- **pop**():从堆栈中弹出一个元素。

例 5-20 给出一个使用类 linkedList < E >实现队列和堆栈功能的 Java 示例代码。

例 5-20　一个使用类 linkedList < E >实现队列和堆栈功能的 Java 示例代码(LinkedListTest.java)

```
 1  import java.util.LinkedList;                              //导入双端队列类 LinkedList
 2  public class LinkedListTest {                             //测试类
 3      public static void main(String[] args) {              //主方法
 4          int n;
 5          //下面演示使用 Integer 型双端队列实现一个队列
 6          LinkedList< Integer> q = new LinkedList<>();      //Integer 型双端队列
 7          for (n = 1; n <= 3; n++)
 8              q.offer( n );                                 //实现一个队列:在队尾添加一个元素
 9          System.out.println( q );                          //显示队列
10          for (n = 1; n <= 3; n++)
11              System.out.print( q.poll() + "," );           //取出并删除队列头的元素
12          System.out.println( "\n" + q + "\n" );            //再次显示队列,此时队列为空
13          //下面演示使用 Integer 型双端队列实现一个堆栈
```

```
14          LinkedList<Integer> s = new LinkedList<>();             //Integer 型双端队列
15          for (n = 1; n <= 3; n++)
16              s.push( n );                           //实现一个堆栈:向栈中压入一个元素
17          System.out.println( s );                   //显示堆栈
18          for (n = 1; n <= 3; n++)
19              System.out.print( s.pop() + "," );     //从栈中弹出一个元素
20          System.out.println( "\n" + s );            //再次显示堆栈,此时堆栈为空
21  } }
```

在 Eclipse 集成开发环境中运行例 5-20 的程序,运行结果如图 5-20 所示。

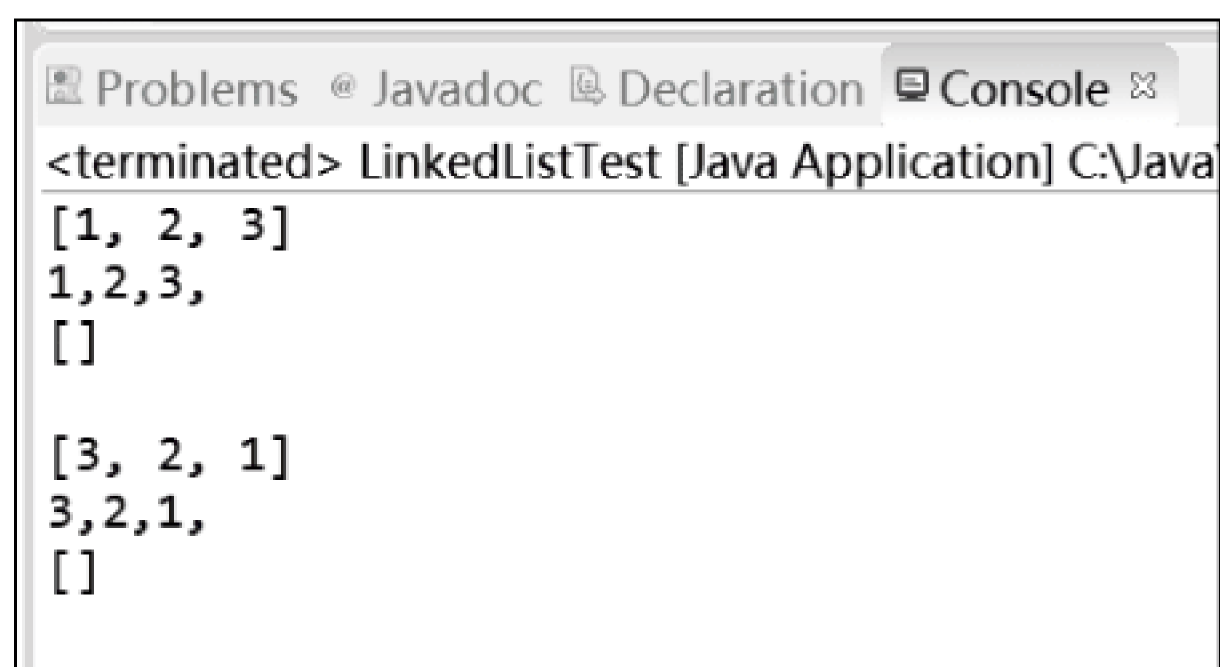


图 5-20　例 5-20 程序的运行结果

3. 集合类 HashSet<E>

Java API 中的集合类 **HashSet**<E>实现了集合接口 Collection<E>。集合类 HashSet<E>具有如下特点。

- 集合中的数据元素是无序的。
- 集合中的数据元素不能重复。**注**:集合类判别数据元素是否重复的方法是,先调用数据元素的 **hashCode**()方法进行快速判重;如果两个元素的哈希码相同,则再调用数据元素的 **equals**()方法进行精确判重(判重就是判断两个元素的内容是否完全一样)。
- 遍历集合中的数据元素,可以使用增强 for 语句和迭代器。**注**:集合中的元素没有序号(下标),不能使用普通 for 语句。

例 5-21 给出一个使用类 HashSet<E>实现集合功能的 Java 示例代码。

例 5-21　一个使用类 HashSet<E>实现集合功能的 Java 示例代码(HashSetTest.java)

```
1  import java.util.HashSet;                                    //导入集合类 HashSet
2  public class HashSetTest {                                   //测试类
3      public static void main(String[] args) {                 //主方法
4          HashSet<String> s = new HashSet<>();                 //String 型集合
5          s.add("1st");  s.add("2nd");  s.add("3rd");          //添加元素
6          s.add("4th");  s.add("5th");
7          System.out.println( s );                   //显示集合,各元素按其哈希码的顺序存放
8          s.remove("4th");                                     //删除元素
9          System.out.println( s );                             //再次显示集合
10         System.out.println( s.size() );                      //显示集合中的元素个数
11 } }
```

在 Eclipse 集成开发环境中运行例 5-21 的程序，运行结果如图 5-21 所示。

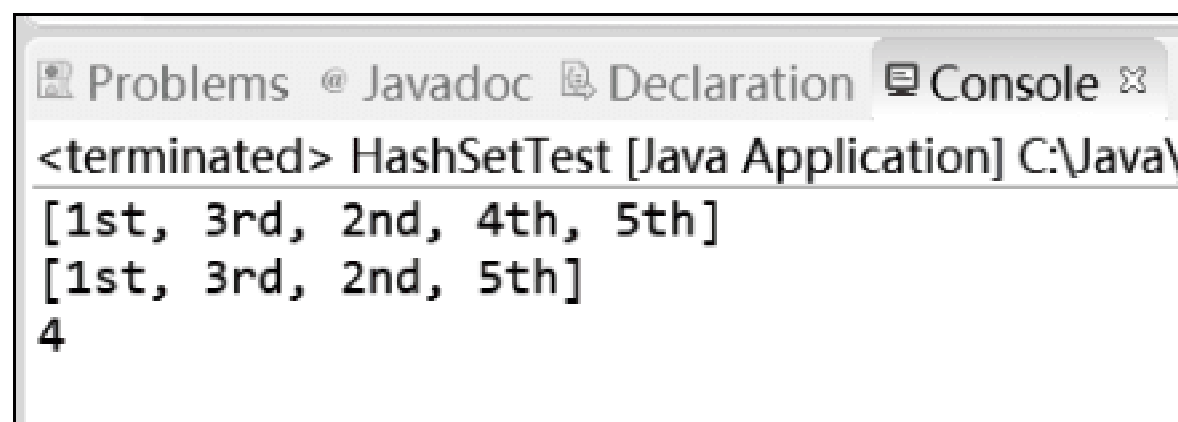


图 5-21　例 5-21 程序的运行结果

4. 映射类 HashMap<K, V>

Java API 中的映射类 **HashMap**<K, V>实现了映射接口 Map<K, V>。使用这个映射类可以存储类似于字典形式的**键值对**(key-value pair)数据。

一个键值对包括两个数据项：一个称为**键**，另一个称为**值**。其含义是将键映射到某个值。例如，学生成绩单中的“张三-92”就是一种“姓名-分数”键值对，英汉字典中的“China-中国”就是一种“英文-中文”键值对。**注**：在程序设计中，映射也可以被称为字典。

使用映射类 HashMap<K, V>所存储的键值对数据集合具有以下特点。

- 映射类中的数据元素是无序的。
- 映射类中数据元素的键不能重复。**注**：映射类判别数据元素的键是否重复的方法是，先调用键所属类的 **hashCode**()方法进行快速判重；如果两个键的哈希码相同，则再调用键的 **equals**()方法进行精确判重。
- 映射类没有迭代器，其中的元素也没有序号(下标)。如需遍历映射，则可将映射或其键转换成集合，再按集合的方式进行遍历。

例 5-22 给出一个使用映射类 HashMap<K, V>存储表 5-4 中学生成绩单的 Java 示例代码。

例 5-22　使用映射类 HashMap<K, V>存储表 5-4 中学生成绩单的 Java 示例代码(HashMapTest.java)

```
1  import java.util.HashMap;                                   //导入映射类 HashMap
2  import java.util.Set;                                       //导入集合类 HashSet
3  public class HashMapTest {                                  //测试类
4      public static void main(String[] args) {                //主方法
5          HashMap<String, Integer> h = new HashMap<>();       //String-Integer 型映射
6          h.put( "张三", 92 );  h.put( "李四", 86 );  h.put( "王五", 95 );    //添加元素
7          //遍历映射:取出键的集合,然后遍历该集合
8          Set<String> kSet = h.keySet();                      //取出映射对象 h 中键的集合
9          for (String k: kSet)  //使用增强 for 语句遍历键的集合 kSet
10         {  System.out.println( k +" - " +h.get(k) );  }//取出映射 h 中键 k 所对应的值
11 } }
```

在 Eclipse 集成开发环境中运行例 5-22 的程序，运行结果如图 5-22 所示。

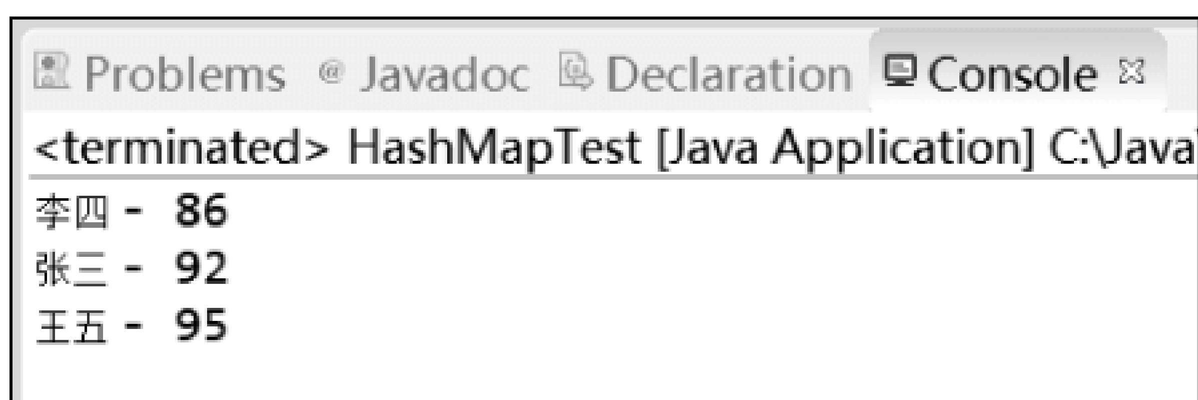


图 5-22　例 5-22 程序的运行结果

5. Java API 数据集合类的继承与实现关系

图 5-23 给出了 Java API 数据集合类的继承与实现关系。

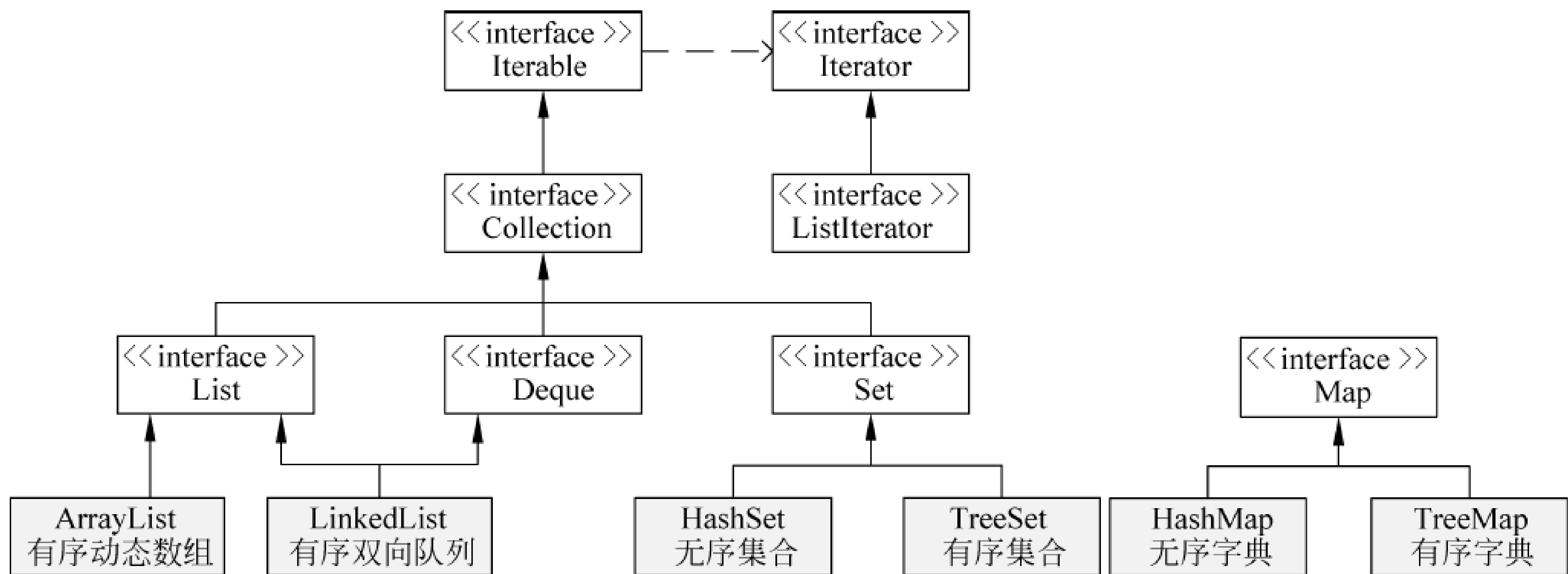


图 5-23　Java API 数据集合类的继承与实现关系

注 1：图 5-23 中，集合类 HashSet 基于 Hash 表存储数据（无序），另外还有一种集合类 TreeSet 则基于红黑树存储数据（自动排序）。这两个类的使用方法是一样的，所不同的是，HashSet 类的内部会通过元素的 hashCode()和 equals()方法来判断集合中是否有重复元素，而 TreeSet 类则是通过元素的 compareTo()方法来进行元素的判重和排序。

注 2：图 5-23 中，映射类 HashMap 基于 Hash 表存储数据（无序），另外还有一种映射类 TreeMap 则基于红黑树存储数据（自动排序）。这两个类的使用方法是一样的，所不同的是，HashMap 类的内部会通过键所属类的 hashCode()和 equals()方法来判断映射中是否有重复元素，而 TreeMap 类则是通过键所属类的 compareTo()方法来进行元素的判重和排序。

这里对本节所讲解的 Java API 数据集合类做一个总结。

(1) 动态数组类 **ArrayList** 可实现动态数组的功能；双端队列类 **LinkedList** 可实现队列或堆栈的功能；集合类 **HashSet** 用于保存无序的数据集合。这 3 个数据集合类都实现了 **Collection**< E >接口，都可以使用增强 for 语句和迭代器遍历集合元素。动态数组类 ArrayList 和双端队列类 LinkedList 是有序集合，可使用普通 for 语句按照元素序号（下标）遍历集合。

(2) 映射类 **HashMap** 用于保存键值对形式的数据集合。映射类实现了 **Map**< K，V >接口。映射类 HashMap 没有迭代器，也没有元素序号（下标）。

(3) 算法类 **Collections** 以静态方法的形式定义了一组专门处理上述数据集合类的算法。

本节习题

1. 下列关于泛型类的描述中,错误的是(　　)。
   A. 带类型参数的类称为泛型类　　B. 类型形参可指代某种具体的数据类型
   C. 使用泛型类时可省略类型实参　　D. 使用泛型类可定义出不同类型的具体类
2. 下列关于泛型的描述中,错误的是(　　)。
   A. 带类型参数的类称为泛型类　　B. 带类型参数的接口称为泛型接口
   C. 带类型参数的方法称为泛型方法　　D. 带类型参数的字段称为泛型字段
3. 下面的类(　　)不是泛型类 G<T>定义出的具体类。
   A. G<Integer>　　B. G<String>　　C. G<Object>　　D. G<double>
4. 下面的类中(　　)没有实现集合接口 Collection<E>。
   A. ArrayList<E>　　B. LinkedList<E>
   C. HashSet<E>　　D. HashMap<K, V>
5. 动态数组类 ArrayList<E>可以实现(　　)的功能。
   A. 动态数组　　B. 堆栈　　C. 无序集合　　D. 字典
6. 双端队列类 LinkedList<E>可以实现(　　)的功能。
   A. 动态数组　　B. 堆栈　　C. 无序集合　　D. 字典
7. 映射类 HashMap<K, V>可以实现(　　)的功能。
   A. 动态数组　　B. 堆栈　　C. 无序集合　　D. 字典
8. 下面的类中(　　)是 Java API 中专门用于处理数据集合的算法类。
   A. ArrayList<E>　　B. LinkedList<E>
   C. Collection<E>　　D. Collections

5.8 枚举类型

和 Java 语言中其他基本数据类型相比,布尔(boolean)类型的主要特点是其值域只有两个取值,即真和假,分别用关键字 true 和 false 表示。实际程序设计任务中也经常会碰到与布尔类型相似的数据,它们的值域是有限的(称为是可枚举的)。例如一个星期只有星期一、星期二、……星期日等 7 天,其值域是可枚举的。Java 语言可以将值域可枚举的数据定义成**枚举类型**。枚举类型值域中的每个取值称为一个**枚举常量**。

注:C 语言中有枚举类型、联合体和结构体等自定义数据类型。Java 语言保留了枚举类型,但没有联合体和结构体。在 Java 语言中,结构体可以用类实现,即结构体实际上是一个只包含公有字段的类。

1. 定义枚举类型

定义枚举类型就是列出该类型所有可能的取值,即枚举常量。枚举常量由程序员命名,习惯上使用全大写字母。定义枚举类型时需使用关键字 **enum**。例如:

```
public enum WeekDay {                                    //定义一个表示星期几的枚举类型 WeekDay
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY      //枚举常量,共 7 个
}
```

2. 使用枚举类型定义变量

可以定义枚举类型的变量,称为**枚举变量**。枚举变量可以保存枚举类型的数据。例如:

```
WeekDay d = WeekDay.MONDAY;          //定义枚举变量 d,初始化为枚举常量 MONDAY(星期一)
```

可以看出,枚举常量便于程序员记忆,所编写的源代码也更容易理解。

3. 枚举类型是一种特殊的类

Java 语言中的枚举类型实际上是一个类,并且都自动继承枚举类 **Enum**<E>。定义枚举类型,实际上是先继承枚举类,然后在此基础上进行扩展,例如添加字段或方法。请读者阅读下面的枚举类 Enum<E>说明文档。

java.lang.**Enum**<E extends Enum<E>>类说明文档

public abstract class **Enum**<E extends Enum<E>>
extends **Object**
implements **Comparable**<E>, **Serializable**

	修　饰　符	类成员(节选)	功 能 说 明
1		String **name**()	返回枚举常量的名字
2		int **ordinal**()	返回枚举常量的序号
3		String **toString**()	将枚举类型转换成字符串

…

Java 编译器在编译枚举类型的类定义代码时会自动添加一个返回枚举常量数组的静态方法 **values**()。例 5-23 给出一个完整的枚举类型 WeekDay 定义及使用示例代码。

例 5-23　一个完整的枚举类型 WeekDay 定义及使用示例代码(EnumTest.java)

```
 1  public class EnumTest {                                //测试类
 2      public static void main(String[ ] args) {          //主方法
 3          for ( WeekDay e: WeekDay.values() )            //列出 WeekDay 中的所有枚举常量
 4              System.out.print( e.name() +", " );        //显示枚举常量的名称
 5          System.out.println();
 6          WeekDay d = WeekDay.MONDAY;                    //定义枚举变量并初始化为 MONDAY
 7          System.out.println( d.ordinal() );             //显示 MONDAY 的内部整数编号
 8          System.out.println( d.name() );                //显示 MONDAY 的名称
 9          d.isWeekend();                                 //检查 MONDAY 是否周末
10  } }
11
12  enum WeekDay {  //定义一个表示星期几的枚举类型 WeekDay
13      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,THURSDAY, FRIDAY, SATURDAY;  //枚举常量
14      public  void isWeekend() {                         //添加方法成员:检查是否周末
15          if (this == SATURDAY || this == SUNDAY)        //枚举类型可以做关系运算
16              System.out.println( "The day is Weekend." );
17          else
18              System.out.println( "The day is not Weekend." );
19  } }
```

在 Eclipse 集成开发环境中运行例 5-23 的程序，运行结果如图 5-24 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> EnumTest (1) [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe
SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
1
MONDAY
The day is not Weekend.
```

图 5-24 例 5-23 程序的运行结果

本节习题

1. C/C++语言中有数组、枚举类型、联合体和结构体等自定义数据类型，Java 语言无法描述的数据类型是(　　)。

A. 数组　　B. 枚举类型
C. 联合体　　D. 结构体

2. 返回枚举类型中枚举常量数组的方法是(　　)。

A. name()　　B. ordinal()
C. values()　　D. toString()

3. 返回枚举类型中枚举常量名称的方法是(　　)。

A. name()　　B. ordinal()
C. values()　　D. toString()

4. 返回枚举类型中枚举常量序号的方法是(　　)。

A. name()　　B. ordinal()
C. values()　　D. toString()

5. 将枚举类型转换成字符串的方法是(　　)。

A. name()　　B. ordinal()
C. values()　　D. toString()

5.9 Java 源程序中的注释和注解

程序员可以对源程序中的代码进行**注释**(comment)，其作用是便于今后自己或其他程序员阅读、理解源代码。Java 语言为程序员提供了 3 种不同的注释形式。

(1) 单行注释。以“//”开头，到所在行的行尾结束。例如：

```
//单行注释内容
```

(2) 多行注释。以“/ * ”开头，以“ * /”结束。例如：

```
/*
    注释内容，可以有多行
 */
```

(3) 文档注释。Java 语言还提供了第 3 种注释形式，这就是**文档注释**(documentation comment)。文档注释以“/ ** ”开头，以“ * /”结束。例如：

```
/ **
    文档注释的内容，可以有多行
 * /
```

5.9.1 文档注释

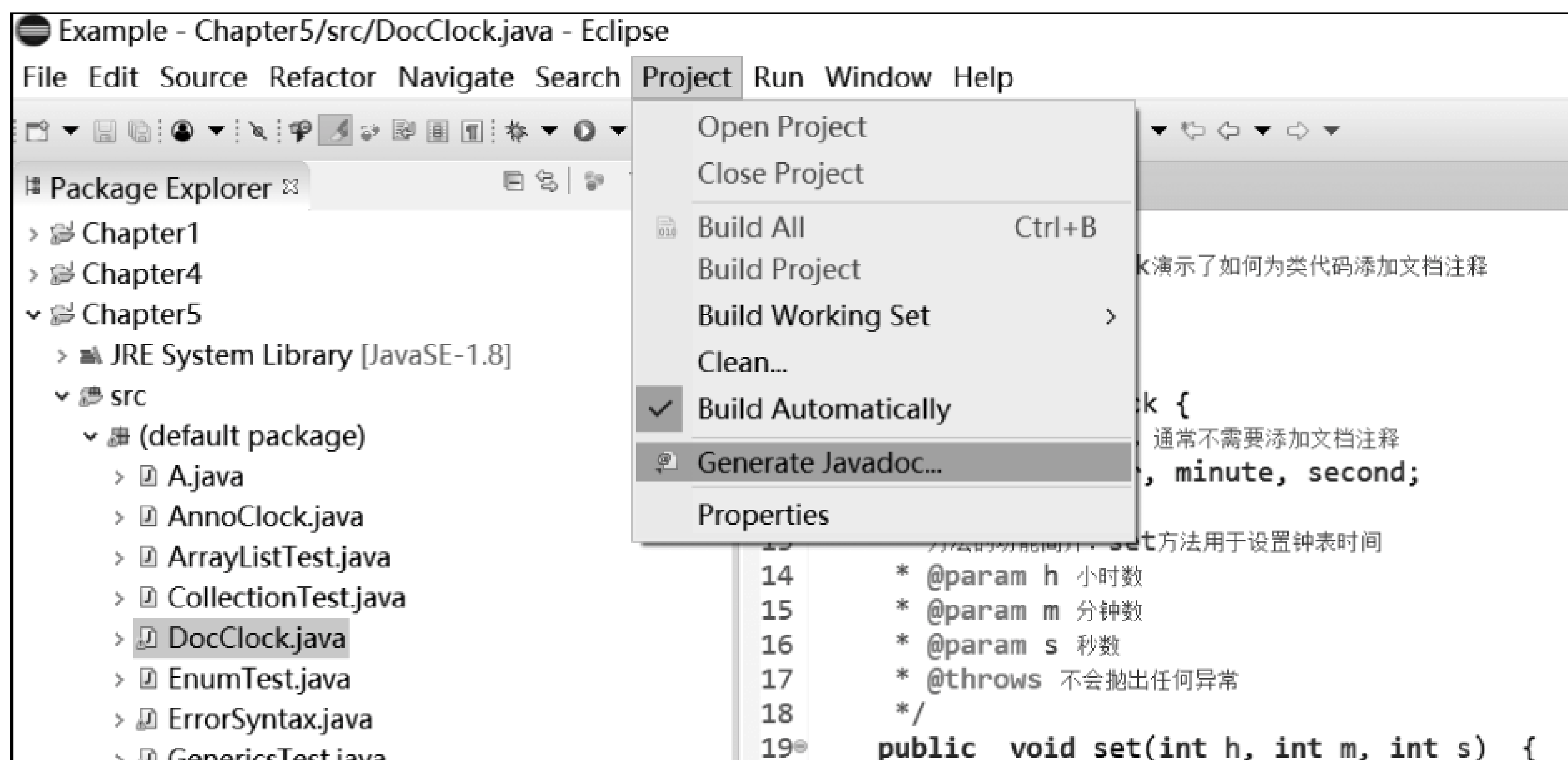
Java 源程序文件(* .java)会被编译成字节码程序文件(* . class)，程序员可以将编译后的字节码程序提供给其他程序员使用。为了让其他程序员了解字节码程序中类的功能和使用方法，程序员需要另外编写一份说明文档。例如，Java API 就提供了全套的说明文档，程序员通过该文档可以详细了解 Java API 的功能和使用方法。

程序员在编写 Java 源程序时，可以为类以及其中的字段、方法添加**文档注释**，然后使用文档生成工具软件(\JDK 安装目录\bin**javadoc**. exe)自动生成说明文档，这样就能大大减轻程序员编写说明文档的工作量。文档注释以“/ ** ”开头，以“ * /”结束。例 5-24 给出一个添加了文档注释的钟表类 DocClock 示例代码。

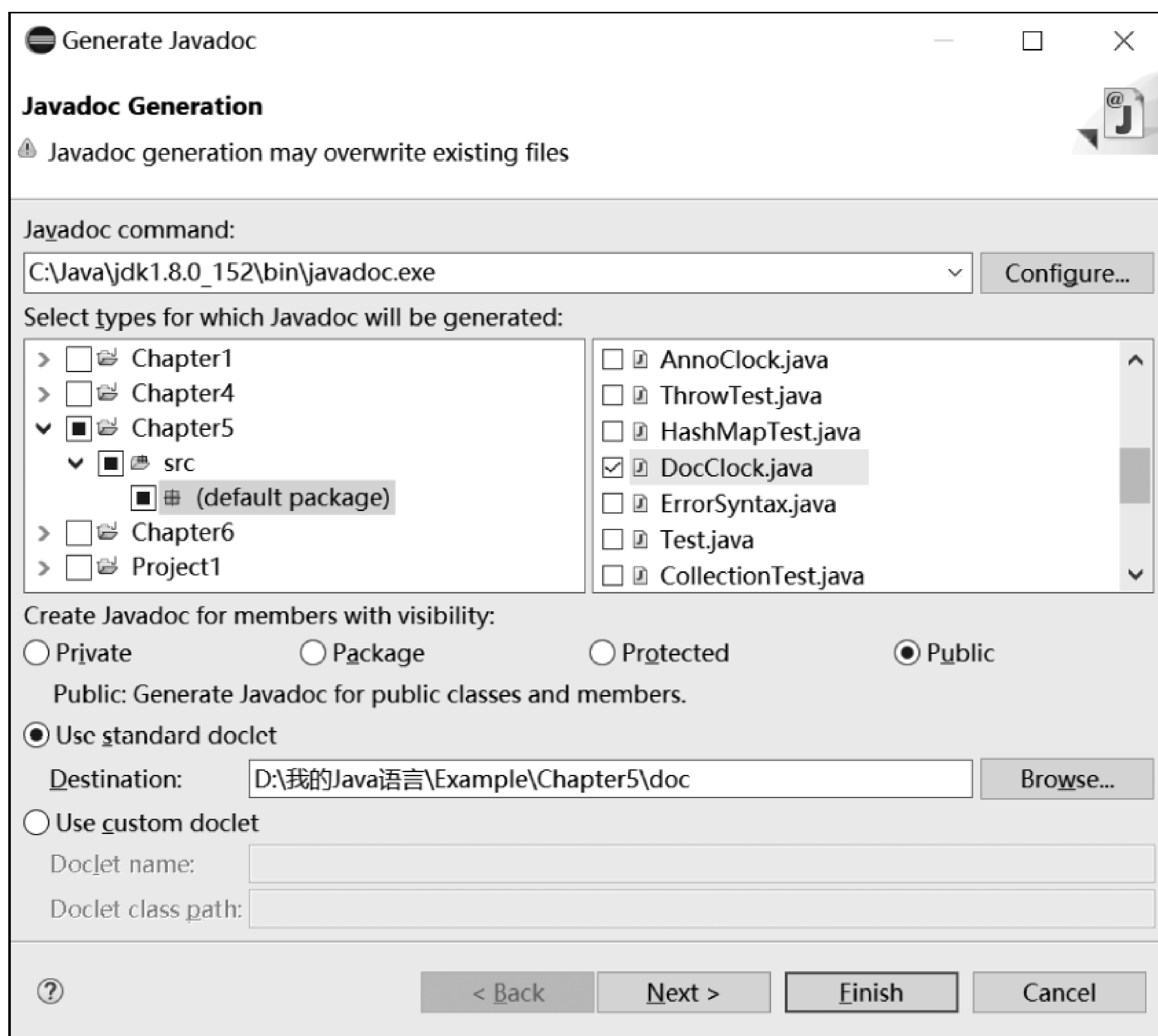
例 5-24 一个添加了文档注释的钟表类 DocClock 示例代码(DocClock. java)

```
1  /**
2   * 类的功能简介:DocClock 是一个钟表类,其中添加了文档注释。
3   */
4  public class DocClock {                        //添加了文档注释的钟表类
5      private int hour, minute, second;      //私有成员不对外开放,通常不需要添加文档注释
6       /**
7        * 方法 set():设置钟表时间,h-小时数,m-分钟数,s-秒数。
8        */
9      public void set(int h, int m, int s)   //添加了文档注释的方法
10     {  hour = h;  minute = m;  second = s;  }
11     public  void show()                    //未添加文档注释的方法
12     {  System.out.println( hour + ":" + minute + ":" + second );  }
13      /**
14       * 方法 toString():将时分秒数据转成字符串格式(重写从 Object 继承来的方法)。
15       */
16     public String toString()               //添加了文档注释的方法
17     {  return String.format("Clock@ %d: %d: %d", hour, minute, second);  }
18      /**
19       * 构造方法:设置钟表时间,h-小时数,m-分钟数,s-秒数。
20       */
21     public DocClock(int h, int m, int s)   //添加了文档注释的方法
22     {  hour = h;  minute = m;  second = s;}
23 }
```

在 Eclipse 集成开发环境中，程序员可以很方便地调用 JDK 的文档生成工具软件 **javadoc**. exe。选择 Project→Generate Javadoc，进入生成 Java 文档对话框，如图 5-25 所示。



(a) 在Eclipse集成开发环境中选择Generate Javadoc菜单



(b) 进入生成Java文档对话框

图 5-25　在 Eclipse 集成开发环境中调用 JDK 的文档生成工具软件

图 5-25 演示了在 Eclipse 集成开发环境中为例 5-24 钟表类 DocClock 自动生成说明文档的操作过程。在图 5-25(b)所示的生成 Java 文档对话框中，程序员需要设置的选项主要有 4 项。

- Javadoc command。在此选项中设置 JDK 文档生成工具软件 javadoc.exe 的安装路径。

- Select types for which Javadoc will be generated。在此选项中选择为哪些 Java 源程序文件生成说明文档。
- Create Javadoc for members with visibility。在此选项中选择为哪种访问权限的类以及类成员生成说明文档。通常应选择 Public，即为公有类以及公有类成员生成说明文档。
- Destination。在此选项中设置所生成说明文档文件的保存路径。

单击图 5-25(b)中的 Finish 按钮，将在 Java 项目根目录下的 doc 子目录中生成一组 HTML 格式的文档说明文件，其中的 index.html 就是文档说明的主页。使用浏览器打开主页文件 index.html，查看所生成的钟表类 DocClock 说明文档，如图 5-26 所示。

程序包 类 使用 树 已过时 索引 帮助

上一个类 下一个类 框架 无框架 所有类

概要：嵌套 | 字段 | 构造器 | 方法 详细资料：字段 | 构造器 | 方法

类 DocClock

java.lang.Object
DocClock

```
public class DocClock
extends java.lang.Object
```

类的功能简介：DocClock是一个钟表类，其中添加了文档注释。

构造器概要

构造器

构造器和说明
DocClock(int h, int m, int s) 构造方法：设置钟表时间，h-小时数，m-分钟数，s-秒数。

(a) 类及构造器（即构造方法）概要部分

方法概要

所有方法 实例方法 具体方法

限定符和类型	方法和说明
void	set(int h, int m, int s) 方法set：设置钟表时间，h-小时数，m-分钟数，s-秒数。
void	show()
java.lang.String	toString() 方法toString：将时分秒数据转成字符串格式（重写从Object继承来的方法）。

从类继承的方法 java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

(b) 方法成员的概要部分

图 5-26 查看为例 5-24 钟表类 DocClock 自动生成的说明文档

JDK 文档生成工具软件会从 Java 源程序文件中提取出如下说明文档内容。

(1) **类的信息**。将具有指定访问权限的类名、类的继承和实现关系、字段成员、方法成员签名等信息提取出来,放入说明文档。

(2) **文档注释**。将所有以"/**"开头、以"*/"结束的文档注释内容提取出来,放入说明文档。

5.9.2 注解

程序员可以定义**注解**(annotation),用于规范文档注释中的某些关键信息,例如程序作者、方法的参数或返回值等说明信息。注解是 Java 语言中一种特殊的接口类型。

1. 注解的定义和使用

Java 语法:**注解的定义和使用**

```
@Documented          //该语句需放在注解定义之前,其作用是指示 Javadoc 识别并提取注解信息
[public] @interface 注解名 {                          //定义注解的语法
    数据类型 注解项 1()  [default 默认值];
    数据类型 注解项 2()  [default 默认值];
    … ;
}

@注解名 (注解项 1 = 注解内容, 注解项 2 = 注解内容, …)    //使用注解的语法
```

语法说明:

- 注解是一种特殊的接口类型,定义时在关键字 **interface** 之前加注解标记符"@"。
- 注解项是以方法成员的形式定义的,定义时可提供默认值。
- 使用注解时需在注解名前加注解标记符"@",并在小括号中给出各注解项的内容。有默认值的注解项可以缺省,缺省时将使用其默认值。
- 如果注解中只有一个注解项,使用时可省略"注解项=",直接给出注解内容。
- 为了指示文档生成工具 Javadoc 识别并提取注解中的信息,要求在注解定义前加"@Documented"进行说明。

例 5-25 给出一个定义和使用注解的钟表类 AnnoClock 示例代码。

例 5-25 一个定义和使用注解的钟表类 AnnoClock 示例代码(AnnoClock.java)

```
1  import java.lang.annotation.*;  //导入定义注解所需的类
2
3  @Documented             //@Document 表示下面的注解 Author 可以被 Javadoc 识别并提取
4  @interface Author {     //定义一个注解 Author,用于生成关于作者信息的说明文档
5      String value();
6  }
7
```

```
 8  @ Documented                      //@Document 表示下面的注解 Info 可以被 Javadoc 识别
 9  @interface Info {                 //定义一个注解 Info,用于生成关于版本和日期的说明文档
10      int version()   default 2;
11      String date()   default "2018/01/01";
12  }
13
14  /**
15   * 类的功能简介:AnnoClock 是一个钟表类,其中同时添加了文档注释和注解.
16   */
17  @ Author( "Kan" )                            //使用注解 Author 来说明类的作者信息
18  @ Info (version = 1, date = "2018/08/20") //使用注解 Info 来说明类的版本和日期
19  public class AnnoClock {                     //同时添加了文档注释和注解的钟表类
20      …                                        //省略部分代码,参见例 5-24
21       /**
22        * 方法 set():设置钟表时间,h-小时数,m-分钟数,s-秒数.
23        */
24      @ Author( "Tom" )
25      public   void set(int h, int m, int s)   //同时添加了文档注释和注解的方法
26      {  hour = h;  minute = m;  second = s;  }
27      public   void show()                     //未添加文档注释或注解的方法
28      {  System.out.println( hour + ":" + minute + ":" + second );  }
29      …                                        //省略部分代码,参见例 5-24
30  }
```

在 Eclipse 集成开发环境中为例 5-25 中的钟表类 DocClock 生成说明文档,然后使用浏览器查看其中的主页文件 index.html,查看结果如图 5-27 所示。

程序包 类 使用 树 已过时 索引 帮助

上一个类 下一个类 框架 无框架 所有类

概要: 嵌套 | 字段 | 构造器 | 方法 详细资料: 字段 | 构造器 | 方法

类 AnnoClock

java.lang.Object
    AnnoClock

```
@Author(value="Kan")
 @Info(version=1,
      date="2018/08/20")
public class AnnoClock
extends java.lang.Object
```

类的功能简介：DocClock是一个钟表类，其中同时添加了文档注释和注解。

(a) 类AnnoClock的说明文档中增加了注解Author和Info信息

图 5-27 查看为例 5-25 钟表类 AnnoClock 自动生成的说明文档

方法详细资料

set

```
@Author(value="Tom")
public void set(int h,
                int m,
                int s)
```

方法set：设置钟表时间，h-小时数，m-分钟数，s-秒数。

show

```
public void show()
```

(b) 方法成员set()的详细资料部分增加了注解Author信息

图 5-27 （续）

2．注解的应用

除了用于生成说明文档之外，注解更主要的用途是在类定义代码中插入附加信息。这些附加信息可以被编译器用于检查语法错误，还可以在运行时向其他程序提供更多关于类的信息。

注解可以有不同的特性，例如注解是否需要被文档生成工具 Javadoc 识别并提取、注解可应用于什么程序元素、注解被保留到什么时候（源代码、字节码或运行时）等。Java 语言使用**元注解**（meta-annotation）来描述这些特性。元注解本身也是一种注解，被定义在 java. lang. annotation 包中。表 5-5 列出了 Java 语言中 5 个常用的元注解。

表 5-5 Java 语言常用的元注解（java. lang. annotation 包）

元 注 解	语 法	说 明
@Documented	@Documented	指定注解需要被文档生成工具 Javadoc 识别并提取
@Repeatable	@Repeatable（注解容器） 举例：定义一个可重复的注解 Schedule @Repeatable(**Schedules**. class) public @interface **Schedule** { … } public @interface **Schedules** { Schedule[] value()； }	指定注解可以对同一程序元素重复使用。编译重复注解时，Java 编译器会将重复的注解保存到一个注解容器中。程序员需要为重复注解另外定义一个注解容器，其中必须包含一个数组类型的注解项 value()
@Target	@Target（元素类型常量） 举例：注解仅用于类里的方法成员 @Target(ElementType. METHOD)	指定注解可应用于什么程序元素。元素类型常量可选择枚举类型 ElementType 中定义的枚举常量：ANNOTATION_TYPE、CONSTRUCTOR、FIELD、LOCAL_VARIABLE、METHOD、PACKAGE、PARAMETER、TYPE 或 TYPE_PARAMETER

续表

元 注 解	语 法	说 明
@Retention	@Retention(注解保留常量) 举例：注解仅保留在源代码中 @Retention(RetentionPolicy. SOURCE)	指定注解被保留到什么时候。注解保留常量可选择枚举类型 RetentionPolicy 中定义的枚举常量 SOURCE、CLASS 或 RUNTIME
@Inherited	@Inherited	指定超类中的注解可以被子类继承

3. Java API 预定义的注解

Java API 还在 java. lang 包中预定义了若干种常用的注解，主要用于向编译器提供附加信息。表 5-6 列出了其中 3 个常用的预定义注解。

表 5-6 Java 语言常用的预定义注解(java. lang 包)

注 解	语 法	说 明
@Override	@Override	表示重新定义超类继承来的方法，即覆盖超类方法。编译器在编译时将检查相关的语法错误，例如方法签名不一致等
@Deprecated	@Deprecated	表示类或类成员是早期(过时)版本，已被弃用，不建议继续使用。如果程序使用了过时版本，编译器在编译时将显示错误提示 (warning)
@SuppressWarnings	@SuppressWarnings(错误列表) 举例：不提示“过时版本”错误 @SuppressWarnings("deprecation")	告知编译器不要显示错误列表中指定的错误提示信息

在例 5-24 中，钟表类 DocClock 重新定义了从超类 Object 继承来的方法 toString()。可以在方法 toString()的定义代码前加上预定义注解“@Override”，明确告知编译器该方法是一个重写的方法。例如：

```
/**
 * 方法 toString():将时、分、秒数据转成字符串格式(重写从 Object 继承来的方法)。
 */
@Override
public String toString()          //同时添加了文档注释和注解的方法
{   return String.format("Clock@%d:%d:%d", hour, minute, second);   }
```

本节习题

1. Java 语言没有形如(　　)的注释形式。

   A. //……　　B. /* …… */　　C. /** …… */　　D. //* …… */

2. 下列注释形式中，(　　)可以被 Java 文档生成工具 Javadoc 自动识别并提取。

   A. //……　　B. /* …… */　　C. /** …… */　　D. //* …… */

3. Java 语言中的注解是一种特殊的(　　)。

A. 类　　B. 接口　　C. 方法成员　　D. 字段成员

4. 使用注解时,注解名前需要添加的字符是(　　)。

A. @　　B. #　　C. *　　D. %

5. 下面的注解中,(　　)表示重写超类继承来的方法。

A. @Override　　B. @Documented

C. @Deprecated　　D. @SuppressWarnings

本章学习要点

- 熟练掌握 Java API 说明文档的阅读方法。
- 学习 Java API 的使用,例如数学类 Math、字符串类 String、基本数据类型的包装类、根类 Object 和系统类 System 等。
- 理解并掌握 Java 语言的 try-catch 异常处理机制。
- 理解泛型编程,并能通过 Java API 中的数据集合类实现动态数组、队列、堆栈、集合和映射等功能。
- 掌握 Java 语言文档注释和注解的基本用法。

本章习题

1. 编写程序。模仿 5.2.1 节的例 5-2,编写一个字符串类 String 的 Java 测试程序。

2. 编写程序。模仿 5.6.2 节例 5-10 的代码结构编写一个求平方根的 Java 程序。要求使用 try-catch 语句处理用户输入负数时的异常。

3. 编写程序。模仿 5.7.4 节的例 5-18,编写一个求学生平均成绩的 Java 程序。要求使用 Java API 中的数组列表类 ArrayList<E>来存储学生成绩。

4. 编写程序。继承 5.9.1 节例 5-24 中的钟表类 DocClock,定义一个手表类 DocWatch,并为其添加文档注释。在 Eclipse 集成开发环境中生成手表类 DocWatch 的说明文档,查看生成的文档说明结果。

第6章 图形用户界面程序

程序执行过程中，通常需要用户输入原始数据或选择功能，这被称为**输入**(input)；程序将计算得到的中间结果或最终结果反馈给用户，这被称为**输出**(output)。用户与程序之间的输入和输出操作统称为**人机交互**(human-computer interaction)。目前，人机交互的形式主要有两种，分别是**命令行界面**(Command Line Interface，CLI)和**图形用户界面**(Graphical User Interface，GUI)。

1. 命令行界面

以前，操作员在控制台上操作计算机，所运行的程序是命令行界面程序(如图 6-1 所示)。控制台主要包括键盘和显示器。操作员通过键盘向计算机下达指令、输入数据，通过显示器查看处理结果。因此人们将键盘称为**标准输入**，将显示器称为**标准输出**，将命令行界面程序称作**控制台**(console)程序。这些称呼一直沿用至今。使用命令行界面的程序需要用户记忆相关的操作命令，这适用于承担系统维护工作的专业技术人员。



图 6-1 在命令行界面中运行的列目录程序 dir

之前所开发的 Java 程序都属于命令行界面程序。例如在 Eclipse 集成开发环境中运行程序时，都是在 Console 标签下输入原始数据，或查看运行结果。

2. 图形用户界面

图形用户界面的程序提供窗口、按钮、菜单等图形操作界面。用户通过指针设备(例如鼠标、触摸屏等)选择程序功能，操作程序，这种操作程序的形式被称为图形用户界面。图6-2就是一个具有图形用户界面的温度换算程序。操作图形用户界面时，用户不需要记忆操作指令，其简单易学，适用于广大的普通用户。本章将学习如何利用Java API来开发具有图形用户界面的程序。

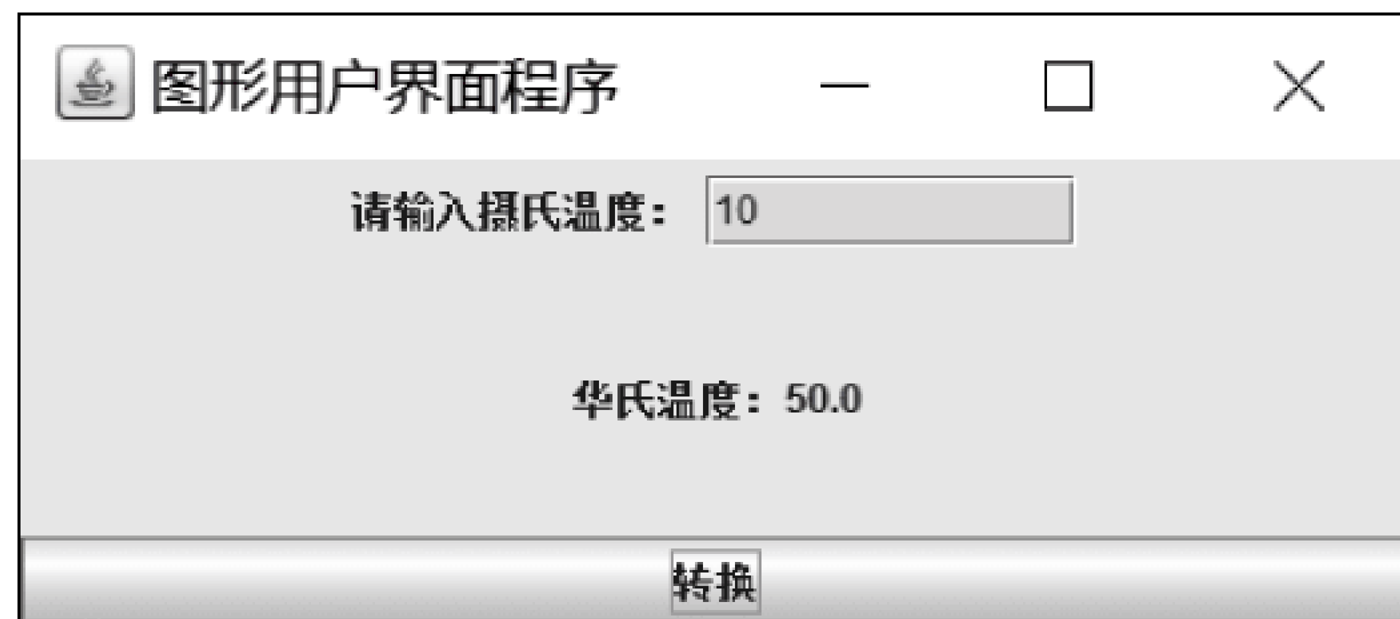


图6-2 具有图形用户界面的温度换算程序

6.1 图形用户界面

为了编写图形用户界面程序，程序员首先需要了解图形用户界面的基本概念和术语。

6.1.1 基本概念和术语

一个图形用户界面就是屏幕(或称为桌面)上的一个程序窗口，如图6-3所示。

1. 屏幕坐标系

计算机**屏幕**(screen)通过屏幕坐标系来确定程序窗口的位置。屏幕坐标系的**原点**为屏幕左上角，坐标单位为**像素**(pixel)。

2. 窗口

窗口(window)由程序创建，用于程序与用户之间的交互。

- **窗口位置**(location)。窗口位置是指窗口左上角在屏幕坐标系中的坐标，以像素为单位。
- **窗口尺寸**(size)。窗口尺寸是指窗口的宽度和高度，以像素为单位。
- **窗口标题**(title)。窗口标题位于窗口的顶部，主要用于显示程序的名称或功能。
- **菜单栏**(menu bar)。菜单栏位于窗口标题的下方。菜单栏通常包含若干**菜单**(menu，或称一级菜单)，每个菜单又会包含若干**菜单项**(menu item，或称子菜单、二级菜单)。图形用户界面程序将程序功能以菜单的形式组织起来，用户通过菜单选择程序功能。一个程序也可以没有菜单，即程序窗口没有菜单栏。
- **内容面板**(content panel)。内容面板是程序窗口的主体，是提供给用户的工作区域。

程序可以在内容面板区域摆放不同的图形组件，这样用户就能通过这些组件输入原始数据、查看处理结果，或选择程序功能。

- **组件**((component)。组件是 Java API 预先定义好的可提供不同功能的图形零件。例如，**按钮**(button)组件可用于选择程序功能，**标签**(label)组件可用于显示文本内容或图片，**文本框**(text field)组件可用于输入文本内容，等等。

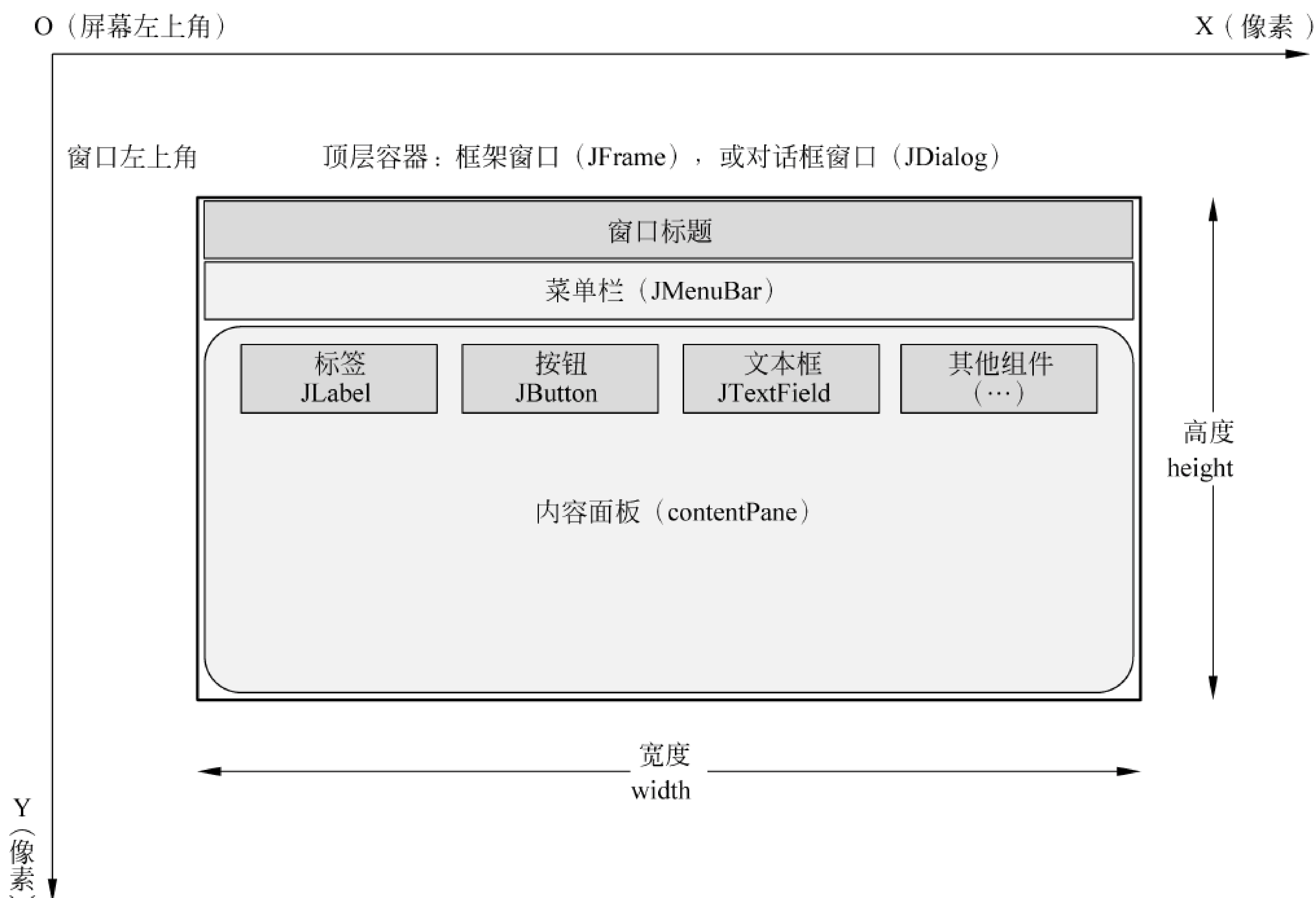


图 6-3　**计算机屏幕上的程序窗口**

3. 容器

在图形用户界面设计中，**容器**(container)是一个专门用于存放其他图形组件的显示区域。例如，窗口就是一个容器，其中可以存放菜单栏和内容面板。内容面板本身也是一个容器，其中可以存放各种不同功能的图形组件。

容器可以作为组件被放入上一层容器中，即容器可以包含子容器。使用子容器的目的是将上层容器的显示区域分割成多个小的显示区域。在 Java API 中，子容器被称为**面板**(panel)，即可以摆放图形组件的面板。程序主窗口是最上层的容器，被称为**顶层容器**(top-level container)。顶层容器不能被放入其他容器。

6.1.2　Java API 中的 swing 包

为了方便程序员编写图形用户界面程序，Java API 将窗口、菜单栏、图形组件和容器等界面元素抽象成数据模型，并定义成一组 Java 类和接口。这些 Java 类和接口被集中放在图形工具包 **javax.swing** 及其下面的子包中。它们共同组成了一个开发图形用户界面程序

的框架,称为 swing 框架。

使用 swing 框架,程序员可以很容易地开发出图形用户界面程序。框架 swing 中的类也称为 swing 类、swing 组件或图形组件。图 6-4 给出了常用的 swing 类及其继承关系图。

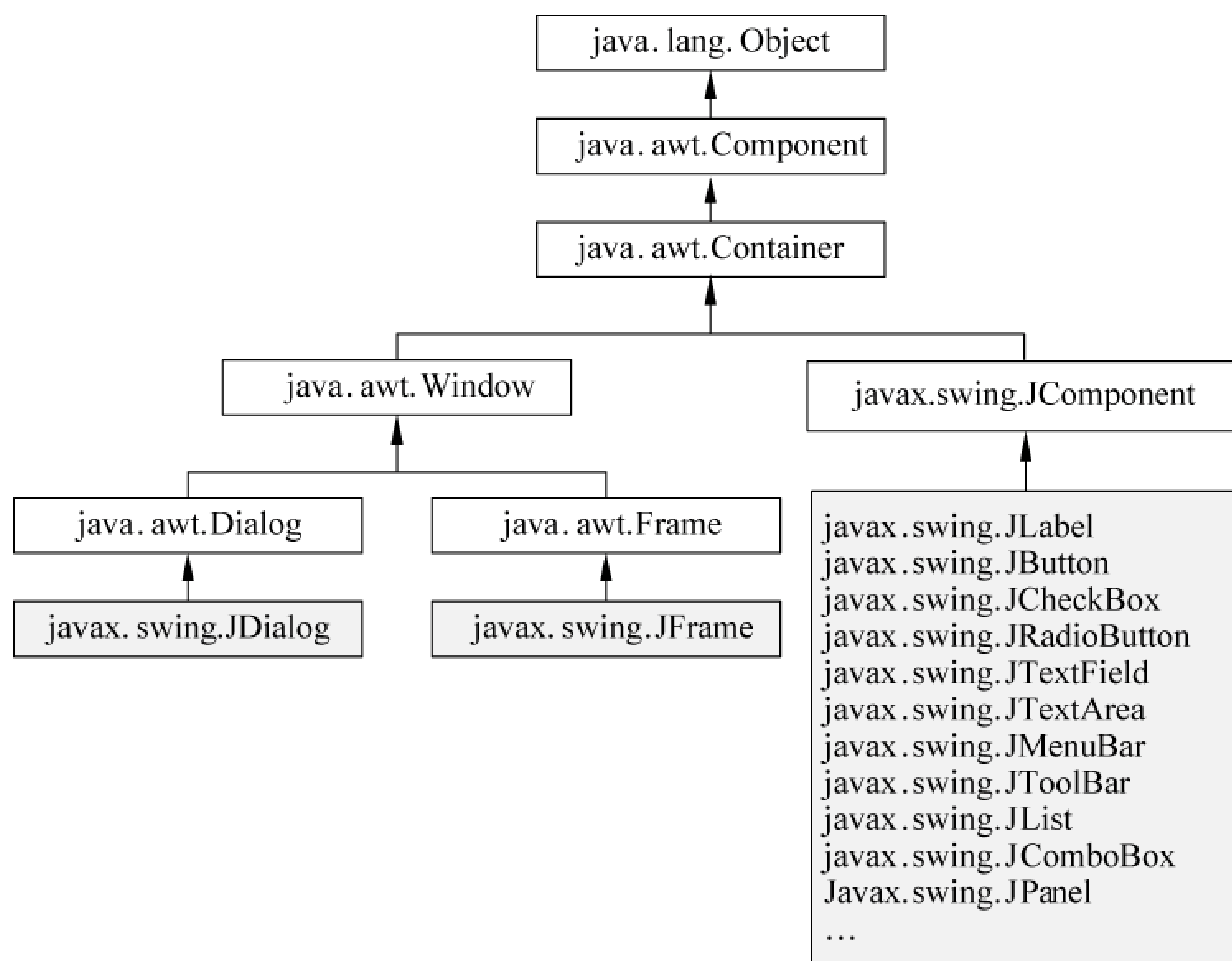


图 6-4 常用的 swing 类及其继承关系图

Java API 最早推出的图形工具包为 awt,即**抽象窗口工具包**(abstract window toolkit)。不久,Java API 又推出了新的图形工具包 swing,它是在 awt 基础上建立起来的一套新的图形用户界面程序框架。使用 Java API 编写图形用户界面程序,通常需要导入以下 3 个包:

```
import java.awt.*;            //导入 java.awt 包中的类
import java.awt.event.*;      //导入 java.awt.event 包中定义的事件类
import javax.swing.*;         //导入 javax.swing 包中的类
```

从图 6-4 可以看出,swing 类构成一个图形类族,其根类为组件类 Component。请读者仔细阅读下面的组件类 Component 说明文档。swing 中的图形组件类都继承了 Component 类中的成员,这些成员很常用。

java.awt.**Component** 类说明文档

public abstract class **Component**
extends **Object**
implements **ImageObserver**, **MenuContainer**, **Serializable**

	修饰符	类成员(节选)	功 能 说 明
1	protected	**Component**()	构造方法
2		void **setSize**(int width, int height)	设置组件尺寸
3		void **setLocation**(int x, int y)	设置组件位置
4		void **setBounds**(int x, int y,int width, int height)	移动或修改组件位置、尺寸

续表

	修饰符	类成员(节选)	功 能 说 明
5		void **setVisible**(boolean b)	显示或隐藏组件
6		void **setEnabled**(boolean b)	设置组件是否可用
7		void **validate**()	验证并重新布局组件
8		void **paint**(Graphics g)	绘制组件
9		void **repaint**()	申请重绘组件
10		void **update**(Graphics g)	刷新并申请重绘组件
11		void **setFont**(Font f)	设置字体
12		void **setCursor**(Cursor cursor)	设置鼠标光标
13		void **setBackground**(Color c)	设置背景颜色
14		void **setForeground**(Color c)	设置前景颜色
15		void **requestFocus**()	申请获得键盘输入焦点
16		boolean **hasFocus**()	检查是否具有键盘输入焦点
17		int **getX**()	获取组件左上角的 x 坐标
18		int **getY**()	获取组件左上角的 y 坐标
19		int **getWidth** ()	获取组件的宽度
20		int **getHeight** ()	获取组件的高度
21		Container **getParent**()	获取所在的容器
22		Graphics **getGraphics**()	获取组件的绘图对象
23		boolean **contains**(int x, int y)	检查某个坐标点是否在本组件的显示区域内
24		void **addKeyListener**(KeyListener l)	添加键盘事件监听器
25		void **addMouseListener**(MouseListener l)	添加鼠标事件监听器
26		void **addMouseMotionListener**(MouseMotionListener l)	添加鼠标移动事件监听器
27		void **addMouseWheelListener**(MouseWheelListener l)	添加鼠标滚轮事件监听器
		…	

本节习题

1. 下列选项中,(　　)不属于人机交互的范畴。
   A. 用户向程序输入数据　　B. 用户选择程序功能
   C. 程序向用户显示结果　　D. 程序执行循环算法
2. 计算机屏幕坐标系的坐标原点是(　　)。
   A. 屏幕左上角　　B. 屏幕右上角　　C. 屏幕左下角　　D. 屏幕右下角
3. 计算机屏幕坐标系的坐标单位是(　　)。
   A. 毫米　　B. 厘米　　C. 英寸　　D. 像素
4. 程序窗口的属性没有包含(　　)。
   A. 窗口位置　　B. 窗口尺寸　　C. 窗口标题　　D. 窗口材质
5. 用于存放其他图形组件的显示区域被称为(　　)。
   A. 按钮　　B. 标签　　C. 文本框　　D. 容器
6. 程序窗口中提供给用户的工作区域称为(　　)。
   A. 标题　　B. 菜单栏　　C. 文本框　　D. 内容面板
7. 下列 Java API 包中,(　　)与 swing 框架无关。

A. java.awt　　B. java.awt.event
C. javax.swing　　D. java.util

8. 组件类 Component 中显示或隐藏组件的方法是(　　)。

A. setSize()　　B. setLocation()
C. setVisible()　　D. setEnabled()

6.2 编写图形用户界面程序

编写一个图形用户界面程序,首先需要创建程序的主窗口。6.1.2 节图 6-4 中的类 **JFrame** 称为**框架窗口**类,它是一个顶层容器。图形用户界面程序通常以类 JFrame 所创建的框架窗口对象作为程序的主窗口。

6.2.1 框架窗口类 JFrame

框架窗口类 JFrame 主要用于创建程序的主窗口。

1. 创建程序窗口

例 6-1 给出一个使用类 JFrame 编写的简单的图形用户界面演示程序。

例 6-1　一个使用框架窗口类 JFrame 编写的图形用户界面演示程序(GUITest.java)

```
 1  import java.awt.*;                           //导入 java.awt 包中的类
 2  import java.awt.event.*;                     //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                        //导入 javax.swing 包中的类
 4
 5  public class GUITest {                                          //主类
 6      public static void main(String[] args) {                    //主方法
 7          JFrame w = new JFrame();             //创建框架窗口类 JFrame 的对象
 8          w.setTitle("图形用户界面演示程序");                     //设置窗口标题
 9          w.setLocation(100, 100);                                //设置窗口位置
10          w.setSize(460, 300);                                    //设置窗口尺寸
11          w.setVisible(true);                                     //设置窗口为可见状态
12          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );     //关闭窗口时退出程序
13  } }
```

在 Eclipse 集成开发环境中运行例 6-1 的程序,运行结果如图 6-5 所示。



图 6-5　例 6-1 程序的运行结果

请读者阅读下面的框架窗口类 JFrame 说明文档。

javax. swing. **JFrame** 类说明文档

public class **JFrame**
extends **Frame**
implements **WindowConstants**，**Accessible**，**RootPaneContainer**

	修饰符	类成员(节选)	功 能 说 明
1	static	int **EXIT_ON_CLOSE**	常量,关闭窗口时退出程序
2		**JFrame**()	构造方法
3		**JFrame**(String title)	构造方法
4		void **setSize**(int width, int height)	设置窗口尺寸
5		void **setLocation**(int x, int y)	设置窗口位置(左上角)
6		void **setTitle**(String title)	设置窗口标题
7		void **setDefaultCloseOperation**(int operation)	设置关闭窗口时的默认操作
8		void **setLayout**(LayoutManager manager)	设置窗口的布局管理器
9		Container **getContentPane**()	取得窗口的内容面板
10		void **setContentPane**(Container contentPane)	设置窗口的内容面板
11		JLayeredPane **getLayeredPane**()	取得窗口的分层面板
12		Component **getGlassPane**()	取得窗口的透明面板
13		JRootPane **getRootPane**()	取得窗口的根面板
14		void **setJMenuBar**(JMenuBar menubar)	设置窗口的菜单栏
15		JMenuBar **getJMenuBar**()	取得窗口的菜单栏
16		void **paint**(Graphics g)	在窗口中绘图
17		void **repaint**()	重绘窗口中的内容
18		void **setVisible**(boolean b)	设置窗口是否为可见状态

…

2. 在窗口中绘图

程序需要在自已的程序窗口中显示文本信息,或绘制图形,这被统称为在窗口中**绘图**(paint)。例 6-2 给出一个在窗口中绘图的 Java 演示程序。

例 6-2　一个在窗口中绘图的 Java 演示程序(HelloWorld. java)

```
 1  import java.awt.*;                          //导入 java.awt 包中的类
 2  import java.awt.event.*;                    //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                       //导入 javax.swing 包中的类
 4
 5  public class HelloWorld {                   //主类
 6      public static void main(String[] args) {   //主方法
 7          JFrame w = new JFrame();            //创建框架窗口类 JFrame 的对象
 8          w.setTitle("图形用户界面演示程序");  //设置窗口标题
 9          w.setSize(460, 300);                //设置窗口尺寸
10          w.setLocation(100, 100);            //设置窗口位置
11          w.setVisible(true);                 //设置窗口为可见状态
12          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );  //关闭窗口时退出程序
```

```
13          //以下为在窗口中绘图的 Java 代码
14          Graphics g = w.getGraphics();                    //获取窗口的绘图对象
15          Font ef = new Font("TimesRoman", Font.PLAIN, 16); //创建字体对象
16          g.setFont( ef );                                 //设置字体
17          g.drawString("Hello, World!", 20, 80);           //显示文字信息
18          Font cf = new Font("楷体", Font.PLAIN, 24);       //创建字体对象
19          g.setFont( cf );                                 //设置字体
20          g.drawString("你好,世界!", 20, 120);              //显示文字信息
21          g.setColor( Color.BLACK );                       //设置填充颜色
22          g.fillRect(20, 150, 100, 100);                   //画一个实心矩形
23          g.setColor( Color.RED );                         //设置绘图颜色
24          g.drawRect(20, 150, 100, 100);     //画一个矩形框,此处是为上面的实心矩形加框
25  } }
```

为了在窗口中绘图,程序员需要做如下两件事情。

(1) 获得窗口的绘图对象。例如:

```
Graphics g = w.getGraphics();                //获取窗口 w 的绘图对象
```

(2) 调用绘图对象的显示文本方法在窗口中显示信息,或调用绘图对象的绘图方法在窗口中绘图。例如:

```
g.drawString("Hello, World!", 20, 80);      //显示文本信息
g.drawRect(20, 150, 100, 100);              //画一个矩形框
```

在 Eclipse 集成开发环境中运行例 6-2 的程序,运行结果如图 6-6 所示。



图 6-6 例 6-2 程序的运行结果

3. 图形类 Graphics

窗口的绘图对象是属于图形类 **Graphics**(实际上是其子类)的对象。图形类 Graphics 包含颜色、字体等绘图属性,还包含显示文本或图像、画直线、画矩形、画椭圆或圆形等绘图方法。图形类 Graphics 是一个抽象类,不同绘图设备(例如显示器或打印机)会继承这个抽象类,并具体实现其中的绘图方法。请读者阅读下面的图形类 Graphics 说明文档。

java.awt.**Graphics** 类说明文档

public abstract class **Graphics**
extends **Object**

	修饰符	类成员(节选)	功 能 说 明
1	protected	**Graphics**()	构造方法
2	abstract	void **setColor**(Color c)	设置颜色
3	abstract	void **setFont**(Font font)	设置字体

续表

	修饰符	类成员(节选)	功能说明
4	abstract	void **drawString**(String str, int x, int y)	显示文本信息(字符串)
5	abstract	void **drawLine**(int x1, int y1, int x2, int y2)	画直线
6		void **drawRect**(int x, int y, int width, int height)	画矩形
7	abstract	void **fillRect**(int x, int y, int width, int height)	填充矩形
8	abstract	void **drawOval**(int x, int y, int width, int height)	画椭圆或圆形
9	abstract	void **fillOval**(int x, int y, int width, int height)	填充椭圆或圆形
10	abstract	boolean **drawImage**(Image img,int x, int y, ImageObserver observer)	显示图像
		…	

绘图过程中可能需要使用颜色类 **Color** 来设置颜色，显示文本信息时还会用到字体类 **Font** 来设置字体和大小。请读者阅读下面的颜色类 Color 和字体类 Font 说明文档。

java.awt.**Color** 类说明文档

public class **Color**
extends **Object**
implements **Paint**, **Serializable**

	修饰符	类成员(节选)	功能说明
1	static	Color **BLACK**	颜色常量，黑色
2	static	Color **WHITE**	颜色常量，白色
3	static	Color **RED**	颜色常量，红色
4	static	Color **GREEN**	颜色常量，绿色
5	static	Color **BLUE**	颜色常量，蓝色
6		**Color**(int r, int g, int b)	构造方法，各颜色分量的数值必须为 0～255
7		**Color**(int r, int g, int b, int a)	构造方法，各颜色分量和 Alpha 分量(透明度)的数值必须为 0～255
8		**Color**(int rgb)	构造方法，rgb 的 3 个低字节为红、绿、蓝分量
9		int **getRed**()	返回颜色的红色分量值
10		int **getGreen**()	返回颜色的绿色分量值
11		int **getBlue**()	返回颜色的蓝色分量值
12		int **getRGB**()	将红、绿、蓝分量合并成一个 int 型整数
		…	

java.awt.**Font** 类说明文档

public class **Font**
extends **Object**
implements **Serializable**

	修饰符	类成员(节选)	功能说明
1	static	int **BOLD**	字体常量，粗体
2	static	int **ITALIC**	字体常量，斜体
3	static	int **PLAIN**	字体常量，正常
4		**Font**(String name, int style, int size)	构造方法

续表

	修饰符	类成员(节选)	功 能 说 明
5	static	Font **getFont**(String nm)	创建指定字体名的对象
6		Font **deriveFont**(int style, float size)	调整字体的风格和大小
		…	

程序运行时,用户可能会改变窗口的大小,或最大/最小化窗口。例 6-2 演示的绘图方法存在这样一个缺陷:当窗口尺寸改变时,窗口中已绘制的内容将会丢失。在窗口中绘图的标准方法是继承框架窗口类 JFrame,重写其绘图方法 **paint**()。

6.2.2 继承并扩展框架窗口类 JFrame

继承并扩展框架窗口类 JFrame 的目的是扩展窗口的功能,例如在窗口中绘图,或添加图形组件。例 6-3 演示了在窗口中绘图的标准方法:首先继承框架窗口类 JFrame,然后重写绘图方法 paint()。

例 6-3 一个继承框架窗口类 JFrame 并重写 paint()方法的 Java 示例程序(HelloWorld1.java)

```
 1  import java.awt.*;                                       //导入 java.awt 包中的类
 2  import java.awt.event.*;                                 //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                                    //导入 javax.swing 包中的类
 4
 5  public class HelloWorld1 {                               //主类
 6      public static void main(String[] args) {             //主方法
 7          MainWnd w = new MainWnd();                       //创建并显示主窗口对象
 8          w.repaint();                                     //调用窗口的重绘方法
 9  } }
10
11  class MainWnd extends JFrame {                    //定义主窗口类:继承并扩展框架窗口类 JFrame
12      public MainWnd() {                            //构造方法:完成初始化窗口的功能
13          setTitle("图形用户界面演示程序");                    //设置窗口标题
14          setSize(460, 300);                                   //设置窗口尺寸
15          setLocation(100, 100);                               //设置窗口位置
16          setVisible(true);                                    //设置窗口为可见状态
17          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); //关闭窗口时退出程序
18      }
19
20      public void paint(Graphics g) {                          //重写绘图方法 paint()
21          super.paint( g );                                    //调用超类的 paint()方法
22          Font ef = new Font("TimesRoman", Font.PLAIN, 16); //创建字体对象
23          g.setFont( ef );                                     //设置字体
24          g.drawString("Hello, World!", 20, 80);               //显示英文信息
25          Font cf = new Font("楷体", Font.PLAIN, 24);          //选择字体(即创建字体)
26          g.setFont( cf );                                     //设置字体
27          g.drawString("你好,世界!", 20, 120);                 //显示中文信息
28          g.setColor( Color.BLACK );                           //设置填充颜色
```

```
29          g.fillRect(20, 150, 100, 100);              //画一个实心矩形
30          g.setColor( Color.RED );                    //设置绘图颜色
31          g.drawRect(20, 150, 100, 100);  //画一个矩形框,此处是为上面的实心矩形加框
32   }  }
```

在 Eclipse 集成开发环境中运行例 6-3 的程序，其运行结果与例 6-2 一样(如图 6-6 所示)。但是，最小化或最大化程序窗口，例 6-3 可以正常显示，而例 6-2 则会丢失内容。为什么会这样呢？这里需要了解一下在窗口中绘图的基本原理。

Java API 将窗口、菜单栏、图形组件和容器等界面元素的共性部分抽象出来形成一个超类，这就是组件类 **Component**(参见 6.1.2 节中的图 6-4)。下面介绍在组件类 Component 及其子类中绘图的基本原理和编程方法。

(1) **绘图目的**。在组件中绘图的目的有两个：一是程序需要在组件中向用户显示信息；二是组件因尺寸改变等原因需重绘内容。

(2) **绘图过程**。Java 虚拟机统一负责绘图操作的调度。当需要在组件中显示信息时，程序员应当调用组件的重绘方法 **repaint**()，其含义是请求 Java 虚拟机重新绘制组件，刷新内容。Java 虚拟机在接收到绘图请求后会调用组件的绘图方法 **paint**()，由该方法具体完成绘图操作。组件因尺寸改变等原因需重绘内容时，Java 虚拟机会自动调用组件的 **paint**()方法。

(3) **重写组件的绘图方法** paint()。如果需要在组件中显示信息，程序员可以继承并扩展组件类(通常是组件类 Component 的某个子类，例如 JFrame)，重写其 paint()方法，在该方法中显示文本信息或绘制图形。**注意**，程序员不要直接调用组件的 paint()方法，而应通过 repaint()方法进行间接调用。

(4) **组件的绘图对象**。Java 虚拟机在调用组件的绘图方法 paint()时，会传递一个图形类 Graphics 的绘图对象。程序员应使用这个绘图对象在组件中进行绘图操作。

6.2.3　在窗口中添加图形组件

Java API 以类的形式为程序员提供了丰富的图形组件。例如：

- javax. swing. **JLabel**，标签。
- javax. swing. **JButton**，按钮。
- javax. swing. **JCheckBox**，复选框。
- javax. swing. **JRadioButton**，单选按钮。
- javax. swing. **JTextField**，单行文本框。
- javax. swing. **JTextArea**，多行文本框。
- javax. swing. **JMenuBar**，菜单栏。
- javax. swing. **JToolBar**，工具栏。
- javax. swing. **JComboBox**，下拉列表框。
- javax. swing. **JList**，列表框。
- javax. swing. **JTable**，二维表格。
- Javax. swing. **JPanel**，子面板(容器)。
- ……

程序员可以根据功能要求在程序窗口中添加图形组件。例 6-4 给出一个在窗口中添加图形组件的 Java 示例程序。

例 6-4 一个在窗口中添加图形组件的 Java 示例程序(JComponentTest.java)

```
 1  import java.awt.*;                              //导入 java.awt 包中的类
 2  import java.awt.event.*;                        //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                           //导入 javax.swing 包中的类
 4
 5  public class JComponentTest {                   //主类
 6      public static void main(String[] args) {    //主方法
 7          MainWnd w = new MainWnd();              //创建并显示主窗口对象
 8  }}
 9
10  class MainWnd  extends JFrame {                 //扩展 JFrame
11      private  JButton bEN, bCN;                  //添加两个按钮字段
12      private  JLabel msg = new JLabel();         //再添加一个标签字段,同时创建对象
13      public MainWnd() {                          //构造方法
14          //初始化窗口
15          setTitle( "图形界面演示程序" );
16          setSize(460, 300);setLocation(100, 100);
17          setVisible(true);
18          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
19          //初始化图形组件
20          bEN = new JButton( "English" );         //创建按钮对象
21          bCN = new JButton( "中文" );
22          msg.setOpaque(true);                    //设置标签背景是否透明:不透明
23          msg.setBackground(Color.YELLOW);        //设置标签背景颜色:黄色
24          msg.setText( "Hello, World!");          //在标签上显示文本信息
25          //将组件添加到窗口的内容面板上
26          Container cp = getContentPane();        //获得窗口的内容面板(容器)
27          cp.setLayout( null );                   //设置容器的布局形式:null-手工布局
28          cp.add( bEN );  cp.add( bCN );  cp.add( msg );      //在容器中添加组件
29          bEN.setBounds(10, 10, 200, 50);         //手工设置各组件的位置和尺寸
30          bCN.setBounds(10, 70, 200, 50);
31          msg.setBounds(10, 150, 200, 80);
32          cp.validate();                          //检查并布局容器里的组件
33  }  }
```

在 Eclipse 集成开发环境中运行例 6-4 的程序,运行结果如图 6-7 所示。

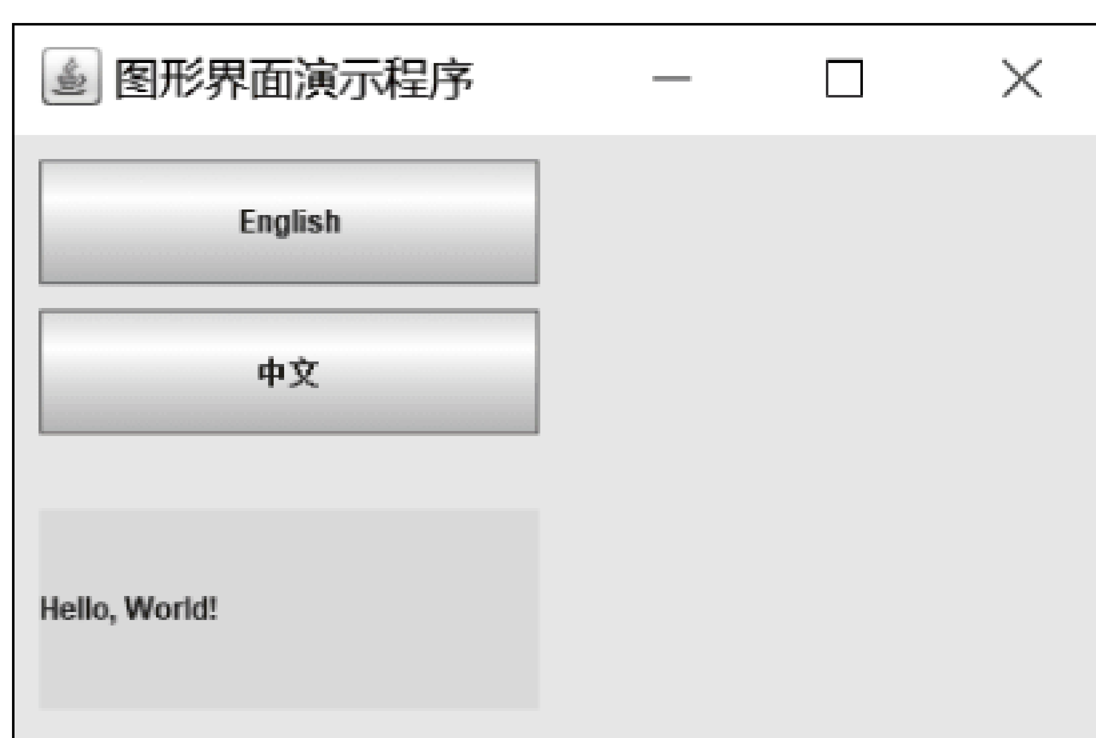


图 6-7 例 6-4 程序的运行结果

图形组件需要放在图形容器中。请读者阅读下面的容器类 Container 说明文档。

java. awt. **Container** 类说明文档

public class **Container**
extends **Component**

	修饰符	类成员(节选)	功 能 说 明
1		**Container**()	构造方法
2		Component **add**(Component comp)	添加组件
3		Component **add**(Component comp, int index)	添加组件并指定其序号
4		void **doLayout**()	对调整后的组件重新布局
5		Component[] **getComponents**()	获取所包含的组件
6		int **getComponentCount**()	获取所包含的组件个数
7		void **remove**(Component comp)	删除组件
8		void **remove**(int index)	删除指定序号的组件
9		void **validate**()	检查并重新布局容器中的组件

…

6.2.4 容器中组件的布局管理

设置容器中图形组件的位置和大小,称为容器的**布局管理**(layout management)。例 6-4 是程序员在编写程序时直接设置各图形组件的位置坐标和尺寸,这种手工布局方法比较烦琐。另外,如果程序运行时用户改变了窗口大小,手工布局不能自动做出适应性调整。

Java API 以类的形式预定义了若干种不同风格的布局管理策略,程序员只要为容器设置某种布局管理策略,容器就能对其中的组件进行自动布局。本节将介绍 4 种常用的布局管理策略,它们分别是流式布局 **FlowLayout**、边框布局 **BorderLayout**、网格布局 **GridLayout** 和卡片式布局 **CardLayout**。

1. 流式布局 FlowLayout

流式布局 FlowLayout 按从左到右的方式对容器中的组件进行排列,当一行排满后自动转入下一行继续排列。例 6-5 给出一个使用流式布局 FlowLayout 的 Java 示例程序。

例 6-5 一个使用流式布局 FlowLayout 的 Java 示例程序(LayoutTest. java)

```
1  import java.awt.*;                          //导入 java.awt 包中的类
2  import java.awt.event.*;                    //导入 java.awt.event 包中定义的事件类
3  import javax.swing.*;                       //导入 javax.swing 包中的类
4
5  public class LayoutTest {                   //主类
6      public static void main(String[] args) {   //主方法
7          JButton btn[] = {                   //创建一个按钮对象数组,包含 9 个按钮
8              new JButton("Button1"), new JButton("Button2"), new JButton("Button3"),
9              new JButton("Button4"), new JButton("Button5"), new JButton("Button6"),
10             new JButton("Button7"), new JButton("Button8"), new JButton("Button9")
11         };
12         JFrame w = new JFrame();            //创建程序窗口
```

```
13          w.setSize(500, 200);  w.setLocation(100, 100); //初始化窗口
14          w.setVisible(true);
15          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
16          //下面演示：设置内容面板的布局策略，然后添加按钮并自动布局
17          w.setTitle( "流式布局 FlowLayout" );            //设置窗口标题
18          Container cp = w.getContentPane();             //获得窗口 w 的内容面板
19          FlowLayout fl = new FlowLayout();              //创建流式布局对象
20          fl.setAlignment(FlowLayout.LEFT);              //设置流式布局为左对齐
21          cp.setLayout( fl );                  //将内容面板(容器)的布局策略设为流式布局
22          for (int n = 0; n < btn.length; n++)           //在内容面板中放入 9 个按钮组件
23              cp.add( btn[n] );
24          cp.validate();                                 //检查并自动布局容器里的组件
25  }  }
```

例 6-5 使用 3 条语句来创建并设置容器的流式布局对象(第 19～21 行)，可将这 3 条语句简写为如下的一条语句：

```
cp.setLayout( new FlowLayout(FlowLayout.LEFT) );          //简写形式
```

在 Eclipse 集成开发环境中运行例 6-5 的程序，流式布局的界面效果如图 6-8 所示。

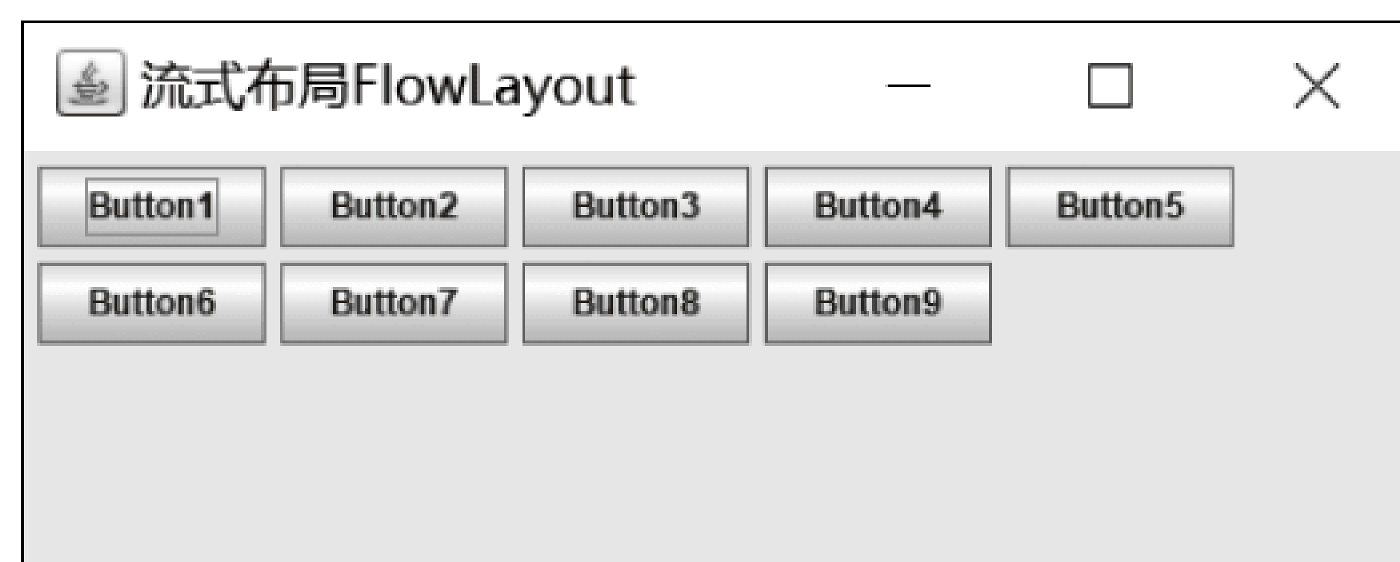


图 6-8 例 6-5 程序的流式布局界面效果

2. 边框布局 BorderLayout

边框布局 BorderLayout 把容器划分为**上**(北，NORTH)、**下**(南，SOUTH)、**左**(西，WEST)、**右**(东，EAST)、**中**(CENTER)5 个区域。如果使用边框布局策略，在向容器中添加组件时需指明添加在哪个区域。每个区域只能存放一个组件。框架窗口的内容面板**默认**使用边框布局策略。

按如下形式修改例 6-5 中的代码第 17～24 行，将流式布局 FlowLayout 改为边框布局 BorderLayout。

```
w.setTitle( "边框布局 BorderLayout" );          //设置窗口标题
Container cp = w.getContentPane();             //获得窗口 w 的内容面板
cp.setLayout( new BorderLayout() );            //将内容面板(容器)的布局策略设为边框布局
cp.add( btn[0], BorderLayout.NORTH );  cp.add( btn[1], BorderLayout.SOUTH );   //添加组件
cp.add( btn[2], BorderLayout.WEST );   cp.add( btn[3], BorderLayout.EAST);
cp.add( btn[4], BorderLayout.CENTER );
cp.validate();                                 //检查并自动布局容器里的组件
```

边框布局的界面效果如图 6-9 所示。**注：**边框布局最多只能添加 5 个组件。

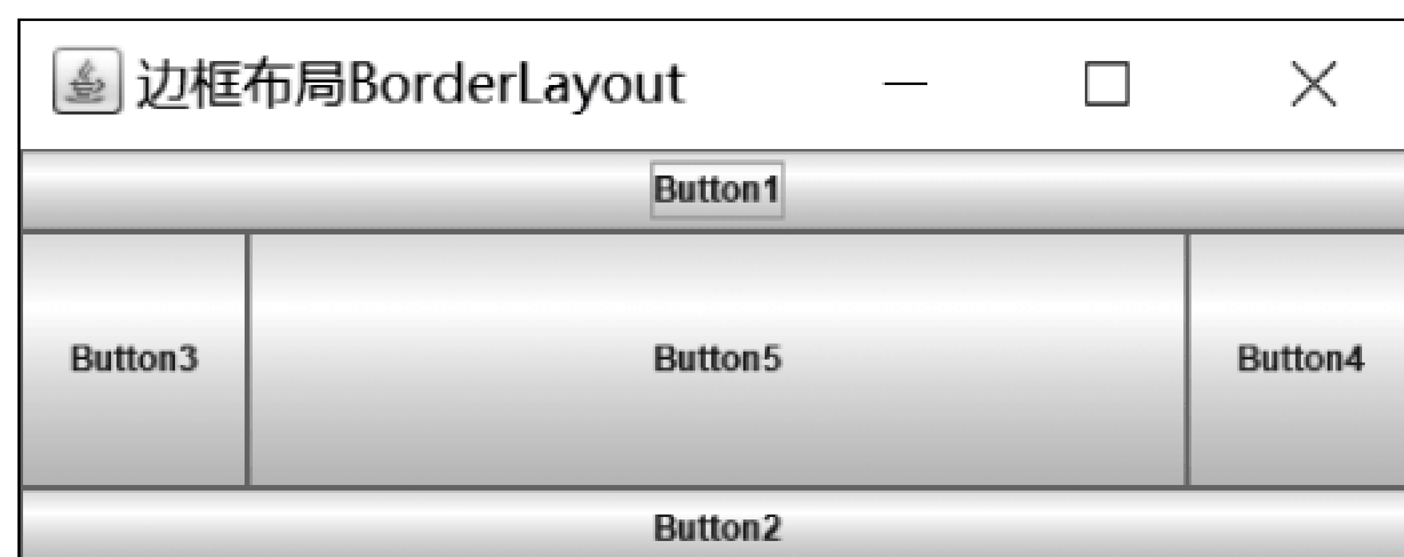


图 6-9　边框布局的界面效果

3. 网格布局 GridLayout

网格布局 GridLayout 将容器均匀地划分成二维网格，网格的大小都相同。添加组件时按"先行后列"的顺序依次放入网格，每个网格摆放一个图形组件。创建网格布局对象时需指明行数和列数。

按如下形式修改例 6-5 中的代码第 17～24 行，将流式布局 FlowLayout 改为网格布局 GridLayout。

```
w.setTitle( "网格布局 GridLayout" );  //设置窗口标题
Container cp = w.getContentPane();    //获得窗口 w 的内容面板
cp.setLayout( new GridLayout(3, 3) ); //将内容面板(容器)的布局策略设为 3 行 3 列的网格布局
for (int n = 0; n < btn.length; n++)  //添加组件
    cp.add( btn[n] );
cp.validate();                        //检查并自动布局容器里的组件
```

网格布局的界面效果如图 6-10 所示。

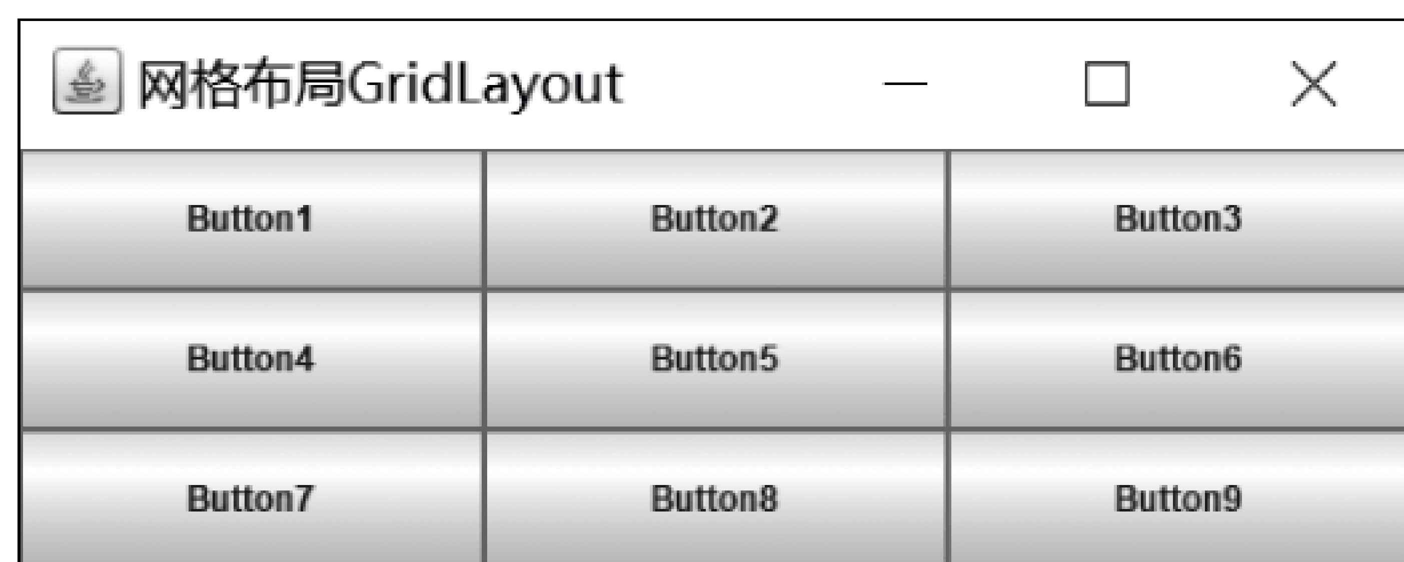


图 6-10　网格布局的界面效果

4. 卡片式布局 CardLayout

卡片式布局 CardLayout 是以层叠方式在容器中放置图形组件的。卡片式布局将多个组件当作是叠加在一起的卡片，每次只能看见最上面的那个组件。

按如下形式修改例 6-5 中的代码第 17～24 行，将流式布局 FlowLayout 改为卡片式布局 CardLayout。

```
w.setTitle( "卡片式布局 CardLayout" );    //设置窗口标题
Container cp = w.getContentPane();        //获得窗口 w 的内容面板
CardLayout cl = new CardLayout();         //创建卡片式布局对象
```

```
cp.setLayout( cl );                    //将内容面板(容器)的布局策略设为卡片式布局
for (int n = 0; n < btn.length; n++)   //添加组件
    cp.add( btn[n] );
cp.validate();                         //检查并自动布局容器里的组件
```

卡片式布局的界面效果如图 6-11 所示。

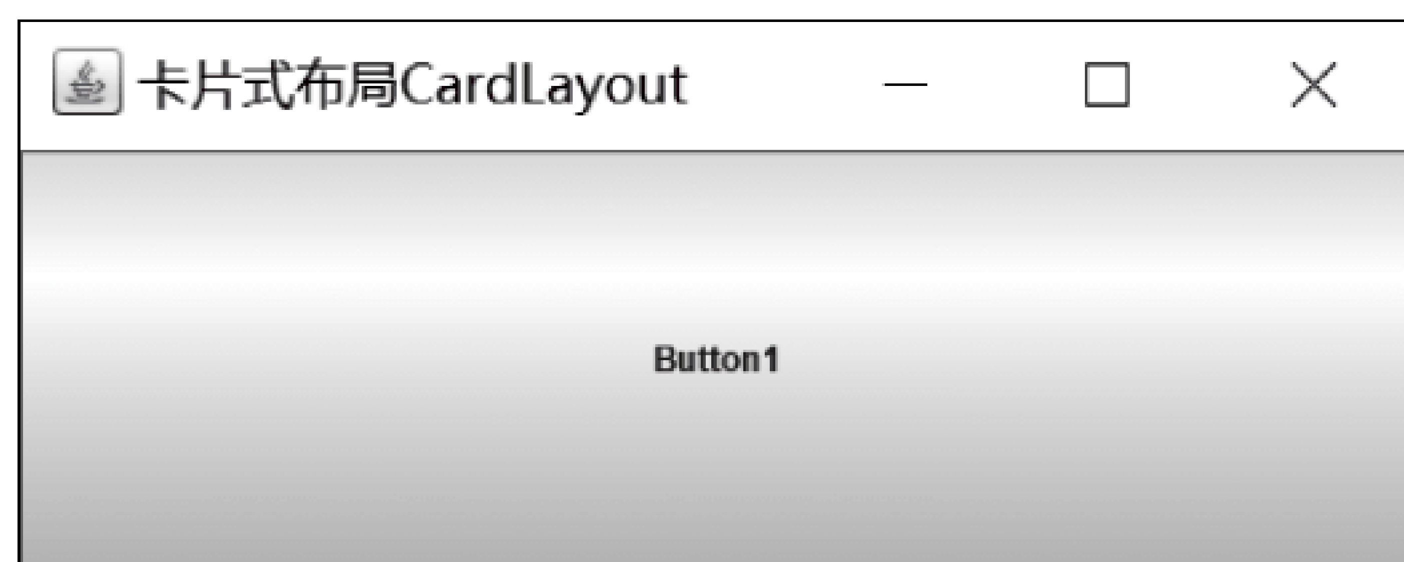


图 6-11 卡片式布局的界面效果

使用卡片式布局策略时，每次只能看见最上面的那个组件，可以使用如下方法来翻阅层叠在一起的组件：

```
cl.first( cp );        //显示第一张卡片：调用卡片式布局对象的 first()方法
cl.next( cp );         //显示下一张卡片：调用卡片式布局对象的 next()方法
cl.previous( cp );     //显示上一张卡片：调用卡片式布局对象的 previous()方法
```

本节习题

1. 下列关于框架窗口类 JFrame 的描述中，错误的是(　　)。
   A. JFrame 主要用于创建程序的主窗口　　B. JFrame 是一个顶层容器
   C. JFrame 中包含一个内容面板　　D. JFrame 不能被继承和扩展
2. 框架窗口类 JFrame 中取得内容面板的方法是(　　)。
   A. getContentPane()　　B. getParent()
   C. getGraphics()　　D. getWidth()
3. 框架窗口类 JFrame 中设置窗口标题的方法是(　　)。
   A. setSize()　　B. setLocation()　　C. setTitle()　　D. setLayout()
4. 图形类 Graphics 中显示文本信息的方法是(　　)。
   A. drawString()　　B. setColor()　　C. setFont()　　D. drawLine()
5. 当需要组件刷新所显示的内容时，程序应当调用组件的(　　)方法。
   A. repaint()　　B. paint()　　C. update()　　D. validate()
6. 容器类 Container 中添加图形组件的方法是(　　)。
   A. add()　　B. remove()
   C. validate()　　D. getComponentCount()
7. 下列布局策略中，(　　)将容器划分成上、下、左、右、中 5 个区域。
   A. FlowLayout　　B. BorderLayout　　C. GridLayout　　D. CardLayout
8. 框架窗口的内容面板，它默认使用的布局策略是(　　)。
   A. FlowLayout　　B. BorderLayout　　C. GridLayout　　D. CardLayout

6.3 响应用户操作

图形用户界面程序提供窗口、按钮、菜单等图形操作界面，用户通过指针设备(例如鼠标、触摸屏等)选择程序功能，操作程序。

本章6.2节已经讲解了如何创建程序窗口，以及如何在窗口中添加组件并对它们进行布局。窗口中的每个组件都代表了某种程序功能。假设用户操作了某个组件，例如单击了某个按钮，程序应当响应用户操作，完成组件所规定的程序功能。本节将具体讲解如何响应用户操作，并最终实现一个具有完整交互功能的图形用户界面程序。

6.3.1 HelloWorld 程序举例

例6-6给出一个HelloWorld程序例子。

例6-6　一个HelloWorld程序例子(JButtonTest.java)

```
1   import java.awt.*;                            //导入 java.awt 包中的类
2   import java.awt.event.*;                      //导入 java.awt.event 包中定义的事件类
3   import javax.swing.*;                         //导入 javax.swing 包中的类
4
5   public class JButtonTest {                    //主类
6       public static void main(String[] args) {  //主方法
7           MainWnd w = new MainWnd();            //创建并显示程序主窗口
8   } }
9
10  class MainWnd  extends JFrame {               //扩展 JFrame
11      private  JButton bEN, bCN;                //添加两个功能按钮
12      private  JLabel msg = new JLabel();       //添加一个信息显示区标签
13      public MainWnd() {                        //构造方法
14          //初始化窗口
15          setTitle( "图形用户界面演示程序" );
16          setSize(460, 300);  setLocation(100, 100);
17          setVisible(true);
18          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
19          //初始化功能按钮,并将功能按钮放入一个子面板(JPanel)
20           bEN = new JButton( "English Button" ); //创建英文按钮
21           bCN = new JButton( "中文按钮" );       //创建中文按钮
22           JPanel bp = new JPanel();            //创建放置按钮的子面板(默认流式布局)
23          bp.add( bEN );       bp.add( bCN );   //将两个按钮放入按钮面板
24          //初始化信息显示区标签
25          msg.setOpaque(true);          //如需设置组件的背景色,首先需将背景设为不透明
26          msg.setBackground(Color.WHITE);                //设置标签的背景色
27          msg.setText( "Information area(信息显示区)");  //设置标签里的文本内容
28          //将按钮面板和信息标签放入窗口的内容面板
29           Container cp = getContentPane();     //获得窗口的内容面板(默认边框布局)
30          cp.add( bp, BorderLayout.NORTH);      //将按钮面板放在内容面板的上部
31          cp.add( msg, BorderLayout.CENTER);    //将信息标签放在内容面板的中间
32          cp.validate();                        //检查并自动布局容器里的组件
33  } }
```

例 6-6 代码第 22～23 行专门创建了一个存放按钮的子面板(JPanel),然后再将该子面板放入内容面板(代码第 30 行)。子面板 JPanel 是一个容器,用于存放其他图形组件。子面板 JPanel 的作用主要体现在以下 3 个方面。

(1) 可以通过子面板将多个图形组件组合成一个整体,这样就能将它们当作一个组件放入内容面板并进行布局。

(2) 子面板可以采用与内容面板不同的布局策略。在 Java API 中,内容面板默认采用边框布局 BorderLayout,子面板 JPanel 默认采用流式布局 FlowLayout。

(3) 子面板可以将内容面板的显示区域分割成多个小的显示区域。每个小区域内部单独布局,互不干扰。每个小区域可以继续通过子面板划分成更小的区域。

在 Eclipse 集成开发环境中运行例 6-6 的程序,运行结果如图 6-12 所示。



图 6-12 例 6-6 程序的运行结果

假设希望用户单击图 6-12 中的"中文按钮",程序能够在按钮下方的信息显示区显示一个中文信息,而单击 English Button 则显示一个英文信息。为了实现这样的功能,需要对例 6-6 做进一步完善,添加响应用户操作的程序代码。

6.3.2 Java 事件响应机制

运行图形用户界面程序,程序将弹出程序窗口,等待用户操作。假设用户操作了某个组件,Java 语言将其称为用户触发了某种**事件**(event),并将用户所操作的组件称作该事件的**事件源**(event source)。Java API 通过定义不同**事件类**来区分用户的不同操作,并为每种事件规定了一个处理该事件的算法接口,该算法接口称为**监听器**(listener)**接口**。

1. Java 事件响应机制工作原理

为了响应用户的操作,Java 图形用户界面程序需要添加**事件响应机制**(见图 6-13)。Java 事件响应机制的工作原理如下。

(1) 为了响应用户对图形组件的操作,程序员需要先定义实现某个监听器接口的类,编写具体的事件处理代码。这种用于事件处理的类称为**监听器类**。例如图 6-13 中实现 ActionListener 接口的监听器类,它可以处理用户单击按钮所触发的 ActionEvent 事件。

(2) 程序员为图形组件**注册**(或称添加)一个**监听器对象**,其目的是预先为图形组件注册一个事件发生时的处理方法。例如,图 6-13 中为"中文按钮"注册了一个处理 ActionEvent



图 6-13　Java 事件响应机制

事件的监听器对象。

（3）当用户操作图形组件触发某个事件时，Java 虚拟机会自动调用该图形组件预先注册好的监听器对象中的处理方法。例如，图 6-13 中用户单击“中文按钮”将触发一个 ActionEvent 事件，此时 Java 虚拟机会自动调用为“中文按钮”预先注册好的监听器对象中的处理 ActionEvent 事件的方法 actionPerformed()。该处理方法会向用户反馈操作结果，即响应用户的操作。

在例 6-6 代码的基础上添加事件响应机制。当用户单击图 6-12 中的“中文按钮”时，程序在按钮下方的信息显示区显示一个中文信息“你好，中国！”，如图 6-14 所示。



图 6-14　用户单击“中文按钮”时显示“你好，中国！”

例 6-7 给出了实现图 6-14 程序功能的示例代码。

例 6-7　在例 6-6 代码基础上添加事件响应机制的 HelloWorld 程序例子(JButtonTest.java)

```
  1~9  ......//第 1~9 行与例 6-6 相同,此处省略
   10  class MainWnd  extends JFrame {                  //扩展 JFrame
   11      private  JButton bEN, bCN;                   //添加两个功能按钮
   12      private  JLabel msg = new JLabel();          //添加一个信息显示区标签
   13      public MainWnd() {                           //构造方法
14~32      …                //第 14~32 行与例 6-6 相同,此处省略
   33
   34        //新添加代码:为"中文按钮"注册一个处理 ActionEvent 事件的监听器对象
   35        bCN.addActionListener( new BcnClicked() );
```

```
36     }
37
38     //新添加代码:定义一个处理 ActionEvent 事件的监听器类 BcnClicked(内部类)
39     class BcnClicked implements ActionListener {      //需实现规定的接口 ActionListener
40         public void actionPerformed(ActionEvent e)    //实现接口抽象方法 actionPerformed
41         {   msg.setText( "你好,中国!");     }           //在信息标签 msg 中显示反馈信息
42     }  }
```

例 6-7 中的类 BcnClicked(第 39～42 行)实现了 ActionEvent 事件所对应的监听器接口 ActionListener,因此类 BcnClicked 就是一个专门处理 ActionEvent 事件的监听器类。**注**:BcnClicked 是定义在 MainWnd 中的一个内部类。

为图形组件添加事件响应机制,就是为组件注册一个监听器对象。例如,为了处理单击“中文按钮”的 ActionEvent 事件,例 6-7 中为“中文按钮”注册一个监听器类 BcnClicked 的对象(代码第 35 行),这样程序就能响应用户单击“中文按钮”的操作了。

2. 简化事件监听器代码

如果一个类继承某个超类或实现了某个接口,并且这个类仅被用于创建一个对象,则可以使用匿名类的语法形式来简化程序代码。例如,可以改用匿名类来实现例 6-7 中监听器类 BcnClicked 的功能,具体实现方法如下。

(1) 删除例 6-7 中第 39～42 行的监听器类 BcnClicked 定义代码。

(2) 使用匿名类,按如下形式改写例 6-7 中第 35 行创建监听器对象的代码。

```
bCN.addActionListener( new ActionListener() {       //匿名类:创建对象时给出类体部分的代码
    public void actionPerformed(ActionEvent e)      //类体部分与类 BcnClicked 相同
    {   msg.setText( "你好,世界!");   }
} );
```

如果一个接口只包含一个抽象方法,则这样的接口称为**功能接口**。例如,处理 ActionEvent 事件的算法接口 ActionListener 就是一个功能接口。如果一个类只实现了一个功能接口,并且没有定义任何其他成员,则该类只包含一个方法成员,这样的类称为**功能类**。例如,例 6-7 中的监听器类 BcnClicked 只实现了功能接口 ActionListener,它就是一个功能类。

以匿名类形式实现的功能类称为**匿名功能类**。可以使用**匿名方法**(或称为 **Lambda** 表达式)的语法形式来简化匿名功能类的代码。例如,可以使用匿名方法来改写前面的匿名类,这样就能进一步简化创建监听器对象的代码。

```
bCN.addActionListener( ( ActionEvent e) -> {  //匿名方法:其中只给出形参和方法体部分的代码
    msg.setText( "你好,世界!");               //方法 actionPerformed()的方法体代码
} );
```

图形用户界面程序经常使用匿名类或匿名方法来简化创建监听器对象的代码。例如,可以使用匿名类再为图 6-14 中的 English Button 按钮 bEN 注册一个处理 ActionEvent 事件的监听器对象。

```
bEN.addActionListener( new ActionListener() {      //匿名类:创建对象时给出类体部分的代码
```

```
        public void actionPerformed(ActionEvent e) {  //实现处理事件的方法 actionPerformed()
            msg.setText( "Hello, World!");            //显示一个英文信息
        }
    } );
```

3. 多个图形组件共用监听器对象

可以让多个图形组件共用一个监听器对象，这样也能简化事件处理代码。例如，可以按如下形式编写事件处理代码，让图 6-14 中的 English Button 按钮 bEN 和“中文按钮”bCN 共用同一个处理 ActionEvent 事件的监听器对象 bl。

```
ActionListener bl = new ActionListener() {          //创建一个匿名监听器类的对象 bl
    public void actionPerformed(ActionEvent e) {    //实现处理事件的方法 actionPerformed()
        if (e.getSource() == bEN)                   //检查事件源是否是 English Button
            msg.setText( "Hello, World!");
        else if (e.getSource() == bCN)              //检查事件源是否是"中文按钮"
            msg.setText( "你好,中国!");
    }
};
bEN.addActionListener( bl );                        //中英文按钮共用同一个监听器对象 bl
bCN.addActionListener( bl );
```

Java 集成开发环境一般会提供专门的界面设计器，帮助程序员设计界面，并能自动生成部分代码。Eclipse 就提供了一个界面设计器插件 Windows Builder，但需要单独下载安装。

6.3.3 常用事件类及其监听器接口

本节汇总 Java API 中常用的事件类，并给出它们的监听器接口说明文档。这些事件类以及监听器接口被定义在 java.awt.event 包中。

表 6-1 常用事件类（java.awt.event 包）

事件类	说　明	监听器接口
ActionEvent	单击按钮、选择菜单、在编辑框按 Enter 键确认输入等操作将触发 ActionEvent 事件	ActionListener
ItemEvent	单击单选按钮、勾选复选框、在下拉列表中选择列表选项等操作将触发 ItemEvent 事件	ItemListener
TextEvent	在编辑框中编辑文本将触发 TextEvent 事件	TextListener
AdjustmentEvent	拖动卷滚条（即滚动条）将触发 AdjustmentEvent 事件	AdjustmentListener
ComponentEvent	改变组件位置或大小、显示或隐藏组件等操作将触发 ComponentEvent 事件	ComponentListener
MouseEvent	移动鼠标、按压鼠标按键等操作将触发 MouseEvent 事件，所有图形组件都有这个事件	MouseListener MouseMotionListener
KeyEvent	按压键盘按键将触发 KeyEvent 事件，所有图形组件都有这个事件	KeyListener

下面给出各事件类的监听器接口说明文档。

java. awt. event. **ActionListener** 接口说明文档

public interface **ActionListener**
extends **EventListener**

	修饰符	接口成员(功能接口)	功 能 说 明
1		void **actionPerformed**(ActionEvent e)	处理 ActionEvent 事件的方法

java. awt. event. **ItemListener** 接口说明文档

public interface **ItemListener**
extends **EventListener**

	修饰符	接口成员(功能接口)	功 能 说 明
1		void **itemStateChanged**(ItemEvent e)	处理 ItemEvent 事件的方法

java. awt. event. **AdjustmentListener** 接口说明文档

public interface **AdjustmentListener**
extends **EventListener**

	修饰符	接口成员(功能接口)	功 能 说 明
1		void **adjustmentValueChanged**(AdjustmentEvent e)	处理 AdjustmentEvent 事件的方法

java. awt. event. **ComponentListener** 接口说明文档

public interface **ComponentListener**
extends **EventListener**

	修饰符	接口成员(全部)	功 能 说 明
1		void **componentResized**(ComponentEvent e)	处理组件大小改变事件的方法
2		void **componentMoved**(ComponentEvent e)	处理组件移动事件的方法
3		void **componentShown**(ComponentEvent e)	处理显示组件事件的方法
4		void **componentHidden**(ComponentEvent e)	处理隐藏组件事件的方法

java. awt. event. **MouseListener** 接口说明文档

public interface **MouseListener**
extends **EventListener**

	修饰符	接口成员(全部)	功 能 说 明
1		void **mouseClicked**(MouseEvent e)	处理单击鼠标按键事件的方法
2		void **mouseEntered**(MouseEvent e)	处理鼠标进入组件事件的方法
3		void **mouseExited**(MouseEvent e)	处理鼠标退出组件事件的方法
4		void **mousePressed**(MouseEvent e)	处理鼠标键被按下事件的方法
5		void **mouseReleased**(MouseEvent e)	处理鼠标键被松开事件的方法

java. awt. event. **MouseMotionListener** 接口说明文档

public interface **MouseMotionListener**
extends **EventListener**

	修饰符	接口成员(全部)	功 能 说 明
1		void **mouseMoved**(MouseEvent e)	处理鼠标移动事件的方法
2		void **mouseDragged**(MouseEvent e)	处理鼠标拖动事件的方法

java. awt. event. **KeyListener** 接口说明文档			
public interface **KeyListener** extends **EventListener**			
	修饰符	接口成员(全部)	功 能 说 明
1		void **keyPressed**(KeyEvent e)	处理按下键盘按键事件的方法
2		void **keyReleased**(KeyEvent e)	处理松开键盘按键事件的方法
3		void **keyTyped**(KeyEvent e)	处理敲击键盘按键事件的方法

本节习题

1. 图形用户界面中的事件(event)是(　　)触发的。
   A. 用户　　B. 程序员
   C. 界面中的图形组件　　D. 程序中的算法代码
2. 处理事件的算法接口被称为(　　)。
   A. listener 接口　　B. collection 接口
   C. map 接口　　D. algorithm 接口
3. 描述事件处理算法的方法被定义在(　　)中。
   A. 监听器类　　B. 集合类　　C. 映射类　　D. 算法类
4. 响应并处理某个图形组件的事件,需要为它注册一个(　　)。
   A. 监听器对象　　B. 集合对象　　C. 映射对象　　D. 算法对象
5. 处理 ActionEvent 事件的监听器接口是(　　)。
   A. ActionListener　　B. ItemListener
   C. MouseListener　　D. KeyListener
6. 监听器接口 ActionListener 中处理 ActionEvent 事件的方法名是(　　)。
   A. actionPerformed　　B. itemStateChanged
   C. mouseClicked　　D. keyPressed
7. Java API 中的事件类及监听器接口被定义在(　　)包中。
   A. java. awt　　B. java. awt. event
   C. javax. swing　　D. java. lang
8. 用户单击按钮会触发(　　)事件。
   A. ActionEvent　　B. ItemEvent
   C. ComponentEvent　　D. KeyEvent

6.4　常用图形组件

本节具体介绍 Java API 在 javax. swing 包中定义的各种图形组件类。这些图形组件类都是从同一个超类 JComponent 继承并扩展而来的(参见 6. 1. 2 节中的图 6-4),因此首先请读者阅读下面的超类 JComponent 说明文档。

javax. swing. **JComponent** 类说明文档			
public abstract class **JComponent** extends **Container** implements **Serializable**			
	修饰符	类成员(节选)	功 能 说 明
1		void **grabFocus**()	申请键盘输入焦点
2		void **setOpaque**(boolean isOpaque)	设置是否不透明,即是否启用背景色
3		void **setBorder**(Border border)	设置边框
4		void **setAutoscrolls**(boolean autoscrolls)	设置是否自动滚动
5		void **setEnabled**(boolean enabled)	设置组件是否可用,若不可用则变灰
…			

6.4.1 按钮类 JButton

图形用户界面程序通常以按钮的形式让用户选择程序功能。程序员需使用按钮类 JButton 创建按钮对象,并将其添加到窗口的内容面板或某个子面板中(参见 6.3.1 节中的例 6-6),然后为按钮对象添加响应 ActionEvent 事件的 ActionListener 监听器对象(参见 6.3.2 节中的例 6-7)。

请读者阅读下面的按钮类 JButton 说明文档。

javax. swing. **JButton** 类说明文档			
public class **JButton** extends **AbstractButton** implements **Accessible**			
	修饰符	类成员(节选)	功 能 说 明
1		**JButton**()	构造方法
2		**JButton**(String text)	构造方法(按钮名称)
3		**JButton**(Icon icon)	构造方法(按钮的图标)
4		**JButton**(String text, Icon icon)	构造方法(名称和图标)
5		void **setText**(String text)	设置按钮名称
6		String **getText**()	读取按钮名称
7		void **setHorizontalTextPosition**(int textPosition)	设置名称的水平对齐方式
8		void **setVerticalTextPosition**(int textPosition)	设置名称的垂直对齐方式
9	protected	void **fireActionPerformed**(ActionEvent event)	触发 ActionEvent 事件
10		void **addActionListener**(ActionListener l)	添加 ActionEvent 监听器
…			

6.4.2 标签类 JLabel

图形用户界面程序通常使用标签(或称为静态文本框)向用户显示文本或图像信息。程序员需使用标签类 JLabel 创建标签对象,并将其添加到窗口的内容面板或某个子面板中(参见 6.3.1 节中的例 6-6)。标签对象不需要添加事件监听器。

请读者阅读下面的标签类 JLabel 说明文档。

javax. swing. **JLabel** 类说明文档

public class **JLabel**
extends **JComponent**
implements **SwingConstants**，**Accessible**

	修饰符	类成员(节选)	功 能 说 明
1		**JLabel**()	构造方法
2		**JLabel**(String text)	构造方法(文本信息)
3		void **setText**(String text)	在标签上显示文本信息
4		String **getText**()	读取标签上的文本信息
5		void **setHorizontalTextPosition**(int textPosition)	设置文本的水平对齐方式
6		void **setVerticalTextPosition**(int textPosition)	设置文本的垂直对齐方式
7		void **setIcon**(Icon icon)	在标签上显示图标(图像)
8		Icon **getIcon**()	读取标签上的图标(图像)

…

6.4.3 文本组件类

图形用户界面程序使用文本编辑框接收用户的键盘输入，输入结果为字符串类型。Java API 为文本编辑框定义了两个类：一个是文本字段类 **JTextField**，用于单行输入；另一个是文本区域类 **JTextArea**，用于编辑多行文本。这两个类是从同一个文本组件类 **JTextComponent** 继承并扩展而来的，因此请读者先阅读下面的文本组件类 JTextComponent 说明文档。

javax. swing. text. **JTextComponent** 类说明文档

public abstract class **JTextComponent**
extends **JComponent**
implements **Scrollable**，**Accessible**

	修饰符	类成员(节选)	功 能 说 明
1		**JTextComponent**()	构造方法
2		String **getText**()	读出文本字符串
3		void **setText**(String t)	设置文本字符串
4		void **setEditable**(boolean b)	设置是否可编辑
5		void **select**(int selectionStart, int selectionEnd)	选中指定范围内的文本
6		void **selectAll**()	选中全部文本
7		String **getSelectedText**()	读出选中的文本
8		void **replaceSelection**(String content)	替换选中的文本
9		void **copy**()	将选中的文本复制到剪贴板
10		void **cut**()	将选中的文本剪切到剪贴板
11		void **paste**()	粘贴剪贴板里的文本

…

1. 文本字段类 JTextField

可以使用文本字段类 JTextField 实现单行文本编辑框的功能。用户在文本编辑框中输入内容,按回车(Enter)键表示结束输入,此时将触发 **ActionEvent** 事件。程序员需使用文本字段类 JTextField 创建单行编辑框对象,并为其添加处理 ActionEvent 事件的 ActionListener 监听器。例 6-8 给出一个文本字段类 JTextField 的 Java 演示程序。

例 6-8 一个文本字段类 JTextField 的 Java 演示程序(JTextFieldTest.java)

```
 1  import java.awt.*;                              //导入 java.awt 包中的类
 2  import java.awt.event.*;                        //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                           //导入 javax.swing 包中的类
 4
 5  public class JTextFieldTest {                   //主类
 6      public static void main(String[] args) {    //主方法
 7          MainWnd w = new MainWnd();              //创建并显示程序主窗口
 8  } }
 9
10  class MainWnd  extends JFrame {                 //扩展 JFrame
11      JTextField tf = new JTextField();           //添加一个单行文本编辑框
12      JLabel msg = new JLabel( "Hello, World!"); //添加一个显示信息的标签
13      public MainWnd() {                          //构造方法
14          setTitle( "图形界面演示程序" );         //初始化窗口
15          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
16          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
17          //设置单行文本编辑框 tf
18          tf.setBackground(Color.YELLOW);                 //设置背景色
19          tf.addActionListener( new ActionListener() {   //添加 ActionEvent 事件监听器
20              public void actionPerformed(ActionEvent e) {
21                  msg.setText( "Hello, " + tf.getText() );    //在信息标签上显示信息
22              }
23          } );
24          //在窗口的内容面板上添加组件 tf 和 msg
25          Container cp = getContentPane();        //获得窗口的内容面板(默认边框布局)
26          cp.add( tf, BorderLayout.NORTH );  cp.add( msg, BorderLayout.CENTER );
27          cp.validate();                          //检查并自动布局容器里的组件
28  } }
```

在 Eclipse 集成开发环境中运行例 6-8 的程序,在编辑框中输入“China”,按 Enter 键,运行结果如图 6-15 所示。



图 6-15 例 6-8 程序的运行结果

请读者阅读下面的文本字段类 JTextField 说明文档。

javax. swing. **JTextField** 类说明文档

public class **JTextField**
extends **JTextComponent**
implements **SwingConstants**

	修饰符	类成员(节选)	功 能 说 明
1		**JTextField**()	构造方法
2		**JTextField**(int columns)	构造方法
3		**JTextField**(String text)	构造方法
4		**JTextField**(String text, int columns)	构造方法
5	protected	void **fireActionPerformed**()	触发 ActionEvent 事件
6		void **addActionListener**(ActionListener l)	添加 ActionEvent 监听器
		…	

2. 文本区域类 JTextArea

可以使用文本区域类 JTextArea 实现多行文本编辑框的功能。在多行文本编辑框中按 Enter 键不会触发 ActionEvent 事件。程序员应另外添加组件(例如按钮),为用户提供操作多行文本编辑框的功能。

通常将多行文本编辑框放入一个滚动面板(**JScrollPane**)。当文本内容超出显示区域时,编辑框将自动显示出卷滚条进行滚动。例 6-9 给出一个文本区域类 JTextArea 的 Java 演示程序。

例 6-9　一个文本区域类 JTextArea 的 Java 演示程序(JTextAreaTest. java)

```
1~9  ......//第 1~9 行与例 6-8 相同,此处省略。注:将主类名改为 JTextAreaTest
 10  class MainWnd  extends JFrame {                //扩展 JFrame
 11      JTextArea ta = new JTextArea(2, 10);       //添加一个 2 行 10 列的文本编辑框
 12      JLabel msg = new JLabel();                 //添加一个显示信息的标签
 13      JButton b = new JButton( "显示文本" );      //添加一个按钮
 14      public MainWnd() {                         //构造方法
 15          setTitle( "图形界面演示程序" );          //初始化窗口
 16          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
 17          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 18          //设置多行文本编辑框 ta
 19          ta.setBackground(Color.YELLOW);        //设置多行文本编辑框的背景色
 20          JScrollPane taScroller = new JScrollPane(ta);//将编辑框放入一个滚动面板
 21          b.addActionListener( new ActionListener() {  //为按钮添加 ActionEvent 事件监听器
 22              public void actionPerformed(ActionEvent e) {
 23                  msg.setText( ta.getText() );   //取出编辑框里的内容,并显示到标签中
 24              }
 25          } );
 26          //在窗口的内容面板上添加滚动面板 taScroller、标签 msg 和按钮 b
 27          Container cp = getContentPane();       //获得窗口的内容面板(默认边框布局)
 28          cp.add(taScroller, BorderLayout.NORTH );
 29          cp.add( msg, BorderLayout.CENTER );  cp.add( b, BorderLayout.SOUTH );
 30          cp.validate();                         //检查并自动布局容器里的组件
 31  }  }
```

在 Eclipse 集成开发环境中运行例 6-9 的程序，在编辑框中输入文本内容，单击“显示文本”按钮，运行结果如图 6-16 所示。



图 6-16　例 6-9 程序的运行结果

请读者阅读下面的文本区域类 JTextArea 说明文档。

javax. swing. **JTextArea** 类说明文档			
public class **JTextArea** extends **JTextComponent**			
	修饰符	类成员(节选)	功 能 说 明
1		**JTextArea**()	构造方法
2		**JTextArea**(int rows, int columns)	构造方法
3		**JTextArea**(String text)	构造方法
4		**JTextArea**(String text, int rows, int columns)	构造方法
5		void **append**(String str)	在末尾追加文本
6		void **insert**(String str, int pos)	在指定位置插入文本
…			

6.4.4 单选按钮类与复选框类

如果程序有一组选项，程序员通常会以单选按钮或复选框的形式让用户进行选择。单选按钮通常为圆形，一组单选按钮只能选中其中的一项，程序员需使用单选按钮类 **JRadioButton** 创建单选按钮对象。复选框通常为方形，在一组复选框中可以同时勾选多项，程序员需使用复选框类 **JCheckBox** 创建复选框对象。

用户单击单选按钮或勾选复选框时会同时触发 **ItemEvent** 事件和 **ActionEvent** 事件。程序员可为单选按钮、复选框添加响应 ItemEvent 事件的 **ItemListener** 监听器，或添加响应 ActionEvent 事件的 **ActionListener** 监听器，二者任选其一。

单选按钮类 JRadioButton 和复选框类 JCheckBox 都是从抽象按钮类 **AbstractButton** 继承并扩展而来的，因此请读者先阅读下面的抽象按钮类 AbstractButton 说明文档。**注**：按钮类 JButton 也是从抽象按钮类 AbstractButton 继承并扩展而来的。

javax. swing. **AbstractButton** 类说明文档			
public abstract class **AbstractButton** extends **JComponent** implements **ItemSelectable**, **SwingConstants**			
	修饰符	类成员(节选)	功 能 说 明
1		**AbstractButton**()	构造方法
2		void **setText**(String text)	设置按钮名称

续表

	修饰符	类成员（节选）	功能说明
3		String **getText**()	读取按钮名称
4		void **setHorizontalTextPosition**(int textPosition)	设置名称的水平对齐方式
5		void **setVerticalTextPosition**(int textPosition)	设置名称的垂直对齐方式
6		void **setEnabled**(boolean b)	设置是否可用（是否可选）
7		void **setSelected**(boolean b)	设置选中状态
8		boolean **isSelected**()	检查是否被选中
9	protected	void **fireActionPerformed**(ActionEvent event)	触发 ActionEvent 事件
10		void **addActionListener**(ActionListener l)	添加 ActionEvent 监听器
11	protected	void **fireItemStateChanged**(ItemEvent event)	触发 ItemEvent 事件
12		void **addItemListener**(ItemListener l)	添加 ItemEvent 监听器
		…	

1. 单选按钮类 JRadioButton

例 6-10 给出一个单选按钮类 JRadioButton 的 Java 演示程序，其功能是根据用户选择（单选）显示不同的文本信息。

例 6-10　一个单选按钮类 JRadioButton 的 Java 演示程序（JRadioButtonTest. java）

```
1～9  ......//第 1～9 行与 6.4.3 节例 6－8 相同,此处省略。注：将主类名改为 JRadioButtonTest
 10  class MainWnd  extends JFrame {                          //扩展 JFrame
 11      JRadioButton cbEN  =  new JRadioButton("英文", true);   //单选按钮:英文
 12      JRadioButton cbCN  =  new JRadioButton("中文");          //单选按钮:中文
 13      JRadioButton cbSH  =  new JRadioButton("上海话");        //单选按钮:上海话
 14      JLabel hello  =  new JLabel("Hello, World!", SwingConstants.CENTER);    //信息标签
 15
 16      public MainWnd() {                                    //构造方法
 17          setTitle( "图形界面演示程序" );                       //初始化窗口
 18          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
 19          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 20          //新建一个按钮面板,放入 3 个单选按钮
 21          JPanel bp  =  new JPanel();                        //创建子面板(默认流式布局)
 22          bp.add(cbEN);  bp.add(cbCN);  bp.add(cbSH);       //添加单选按钮组件
 23          //将 3 个互斥的单选按钮合成一组,同时只会有一个被选中
 24          ButtonGroup group  =  new ButtonGroup();           //创建组对象
 25          group.add(cbEN);  group.add(cbCN);  group.add(cbSH);  //加入组中
 26          //在主窗口的内容面板中放入按钮面板 bp 和标签 hello
 27          Container cp  =  getContentPane();          //获得窗口的内容面板(默认边框布局)
 28          cp.add( bp, BorderLayout.NORTH );           //添加按钮面板 bp
 29          cp.add( hello, BorderLayout.CENTER );       //添加信息标签 hello
 30          cp.validate();                              //检查并自动布局容器里的组件
 31          //处理 ItemEvent 事件的监听器:根据单选按钮状态来显示对应的信息
 32          ItemListener il  =  new ItemListener() { //匿名类
 33              public void itemStateChanged( ItemEvent e) {   //处理 ItemEvent 事件的方法
 34                  String msg  =  null;
 35                  if ( cbEN.isSelected() )  msg  =  "Hello, World!";
 36                  else if ( cbCN.isSelected() )  msg  =  "你好,世界!";
```

```
37              else if ( cbSH.isSelected() )  msg = "侬好,世界!";
38              hello.setText( msg );
39          } };
40          //3 个单选按钮对象共用同一个监听器对象 il
41          cbEN.addItemListener(il);  cbCN.addItemListener(il);  cbSH.addItemListener(il);
42  } }
```

在 Eclipse 集成开发环境中运行例 6-10 的程序,运行结果如图 6-17 所示。



图 6-17　例 6-10 程序的运行结果

2. 复选框类 JCheckBox

例 6-11 给出一个复选框类 JCheckBox 的 Java 演示程序,其功能是根据用户选择(可多选)显示不同的文本信息。

例 6-11　一个复选框类 JCheckBox 的 Java 演示程序(JCheckBoxTest.java)

```
1~9  ......//第 1~9 行与 6.4.3 节例 6-8 相同,此处省略。注: 将主类名改为 JCheckBoxTest
10  class MainWnd  extends JFrame {                          //扩展 JFrame
11      JCheckBox cbEN = new JCheckBox("英文");              //复选框:英文
12        JCheckBox cbCN = new JCheckBox("中文");            //复选框:中文
13      JCheckBox cbSH = new JCheckBox("上海话");            //复选框:上海话
14      JLabel hello = new JLabel("", SwingConstants.CENTER);  //信息标签
15
16      public MainWnd() {                                   //构造方法
17          setTitle( "图形界面演示程序" );a                  //初始化窗口
18          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
19          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
20          //新建一个复选框面板,放入 3 个复选框
21          JPanel bp = new JPanel();                        //创建子面板(默认流式布局)
22          bp.add(cbEN);  bp.add(cbCN);  bp.add(cbSH);      //添加复选框组件
23          //在主窗口的内容面板中放入复选框面板 bp 和标签 hello
24          Container cp = getContentPane();         //获得窗口的内容面板(默认边框布局)
25          cp.add( bp, BorderLayout.NORTH );                //添加复选框面板 bp
26          cp.add( hello, BorderLayout.CENTER );            //添加信息标签 hello
27          cp.validate();                          //检查并自动布局容器里的组件
28          //处理 ItemEvent 事件的监听器:根据复选框状态来显示对应的信息
29          ItemListener il = new ItemListener() {           //匿名类
30              public void itemStateChanged(ItemEvent e) { //处理 ItemEvent 事件的方法
31                  String msg = "";
32                  if ( cbEN.isSelected() )  msg += "Hello, World! ";
33                  if ( cbCN.isSelected() )  msg += "你好,世界! ";
34                  if ( cbSH.isSelected() )  msg += "侬好,世界! ";
35                  if ( msg.equals("") )  hello.setText("没有勾选项!");
36                  else  hello.setText( msg );
37          } };
38          //3 个复选框对象共用一个事件监听器
39          cbEN.addItemListener(il);  cbCN.addItemListener(il);  cbSH.addItemListener(il);
40  } }
```

在 Eclipse 集成开发环境中运行例 6-11 的程序,运行结果如图 6-18 所示。



图 6-18　例 6-11 程序的运行结果

6.4.5　列表类

图形用户界面也会以列表或下拉列表的形式让用户在一组选项中进行选择。列表、下拉列表比单选按钮或复选框节省屏幕空间。程序员使用列表类 **JList** < E >创建列表对象,使用下拉列表类 **JComboBox** < E >创建下拉列表对象。

列表类、下拉列表类采用泛型编程,其中的列表选项可以是任意引用类型。程序通常使用字符串形式描述列表选项,例如 JList < **String** >、JComboBox < **String** >。

列表类 JList < E >以列表格式接收用户输入,这时列表类 JList < E >被当作**输入组件**使用。列表类 JList < E >也可以被当作**输出组件**使用,即以列表格式向用户输出信息。输出组件通常不需要响应事件。另一种常用的输出组件是表格类 **JTable**。

1. 下拉列表类 JComboBox < E >

用户选择下拉列表中的选项,这会同时触发 **ItemEvent** 事件和 **ActionEvent** 事件。程序员可为下拉列表添加响应 ItemEvent 事件的 **ItemListener** 监听器,或添加响应 ActionEvent 事件的 **ActionListener** 监听器,二者任选其一。

例 6-12　一个下拉列表类 JComboBox < E >的 Java 演示程序(JComboBoxTest. java)

```
1~9  ......//第 1~9 行与 6.4.3 节例 6-8 相同,此处省略。注:将主类名改为 JComboBoxTest
10   class MainWnd  extends JFrame {                    //扩展 JFrame
11       JComboBox < String > list;                     //字符串型下拉列表
12       String listItems[] = { "英文", "中文", "上海话", "广东话", "闽南话" };  //选项
13       JLabel info = new JLabel("", SwingConstants.CENTER);   //信息标签
14
15       public MainWnd() {                                     //构造方法
16           setTitle( "图形界面演示程序" );                    //初始化窗口
17           setSize(300, 200);  setLocation(100, 100);  setVisible(true);
18           setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
19           //设置下拉列表 list
20           list = new JComboBox < String >( listItems );     //创建下拉列表并初始化选项
21           list.setMaximumRowCount( 3 );                      //设置下拉行数
22           list.setSelectedIndex( 1 );          //设置初始选中的列表选项(编号从 0 开始)
23            //在窗口的内容面板上添加组件
24           Container cp = getContentPane();      //获得窗口的内容面板(默认边框布局)
25           cp.add( list, BorderLayout.NORTH );   //将下拉列表添加到主窗口
26           cp.add( info, BorderLayout.SOUTH );   //将信息显示标签添加到主窗口
27           cp.validate();//检查并自动布局容器里的组件
28           //处理 ItemEvent 事件的监听器:根据下拉列表选中的选项来显示对应的信息
29           ItemListener il = new ItemListener() {              //匿名类
30               public void itemStateChanged(ItemEvent e) {   //处理 ItemEvent 事件的方法
```

```
31
                JComboBox cb = (JComboBox)e.getSource();//获取事件源
32                  String item = ( String)cb.getSelectedItem();  //获取被选中的选项
33                  info.setText( item + " 被选中!" );
34              } };
35              list.addItemListener(il);//为下拉列表添加 ItemListener 监听器
36      } }
```

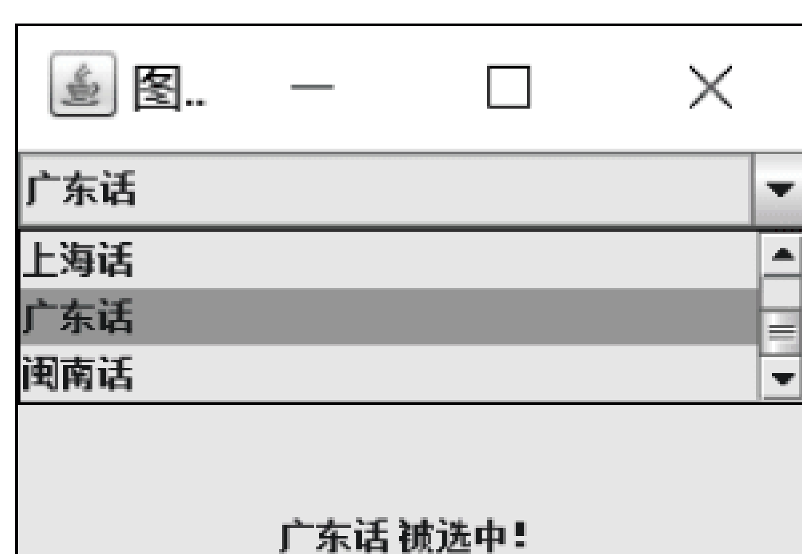


图 6-19 例 6-12 程序的运行结果

在 Eclipse 集成开发环境中运行例 6-12 的程序，运行结果如图 6-19 所示。

2. 列表类 JList<E>

用户选择列表里的选项时会触发 **ListSelectionEvent** 事件，程序员需为列表添加响应这个事件的 **ListSelectionListener** 监听器。

之前用到的事件类都是 awt 定义的，而列表类 JList<E>用到的事件类 ListSelectionEvent 则是 swing 新增加的。处理 ListSelectionEvent 事件需导入 javax.swing.event 包中定义的事件类。例 6-13 给出一个列表类 JList<E>的 Java 演示程序。

例 6-13 一个列表类 JList<E>的 Java 演示程序(JListTest.java)

```
 1  import java.awt.*;                    //导入 java.awt 包中定义的类
 2  //import java.awt.event.*;            //本例不需要导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                 //导入 javax.swing 包中定义的类
 4  import javax.swing.event.*;           //导入 javax.swing.event 包中定义的事件类
 5
 6  public class JListTest {                                    //主类
 7      public static void main(String[] args) {                //主方法
 8          MainWnd w = new MainWnd();                          //创建并显示主窗口对象
 9  } }
10
11  class MainWnd  extends JFrame {                             //扩展 JFrame
12      JList<String> list;                                     //字符串型列表
13      String listItems[] = { "英文", "中文", "上海话", "广东话", "闽南话" };  //选项
14      JLabel info = new JLabel("", SwingConstants.CENTER);    //信息标签
15      public MainWnd() {                                      //构造方法
16          setTitle( "图形界面演示程序" );                      //初始化窗口
17          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
18          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
19          //设置列表 list
20          list = new JList<String>( listItems );        //创建列表并初始化列表选项
21          list.setSelectionMode( ListSelectionModel.SINGLE_SELECTION );  //单选或多选
22          list.setLayoutOrientation( JList.VERTICAL );  //设置纵向或横向布局
23          list.setVisibleRowCount( 3 );                 //设置行数
24          list.setSelectedIndex( 1 );          //设置初始选中的列表选项(编号从 0 开始)
```

```
25          //如果列表选项比较多,需将列表放入一个卷滚面板
26          JScrollPane listScroller = new JScrollPane( list);
27          //在窗口的内容面板上添加组件
28          Container cp = getContentPane();          //获得窗口的内容面板(默认边框布局)
29          cp.add( listScroller, BorderLayout.NORTH ); //将列表卷滚面板添加到主窗口
30          cp.add( info, BorderLayout.CENTER );       //将信息显示标签添加到主窗口
31          cp.validate();                             //检查并自动布局容器里的组件
32          //处理 ListSelectionEvent 事件的监听器:根据列表选中的选项来显示对应的信息
33          ListSelectionListener  lsl = new ListSelectionListener () {  //匿名类
34              public void valueChanged(ListSelectionEvent e) {  //处理列表选择事件
35                  int index = list.getSelectedIndex();          //获取被选中选项的序号
36                  if (index == -1)  info.setText( "无" );       //没有选项被选中
37                  else  info.setText( listItems[ index] + " 被选中!" );  //有选项被选中
38              } };
39              list.addListSelectionListener( lsl );     //为列表 list 添加监听器对象 lsl
40  } }
```

在 Eclipse 集成开发环境中运行例 6-13 的程序,运行结果如图 6-20 所示。

列表类 JList<E>也可以被当作输出组件使用,即以列表格式向用户输出信息。将列表类 JList<E>当作输出组件时的程序代码与例 6-13 基本相同,唯一不同的是输出组件通常不需要响应事件。

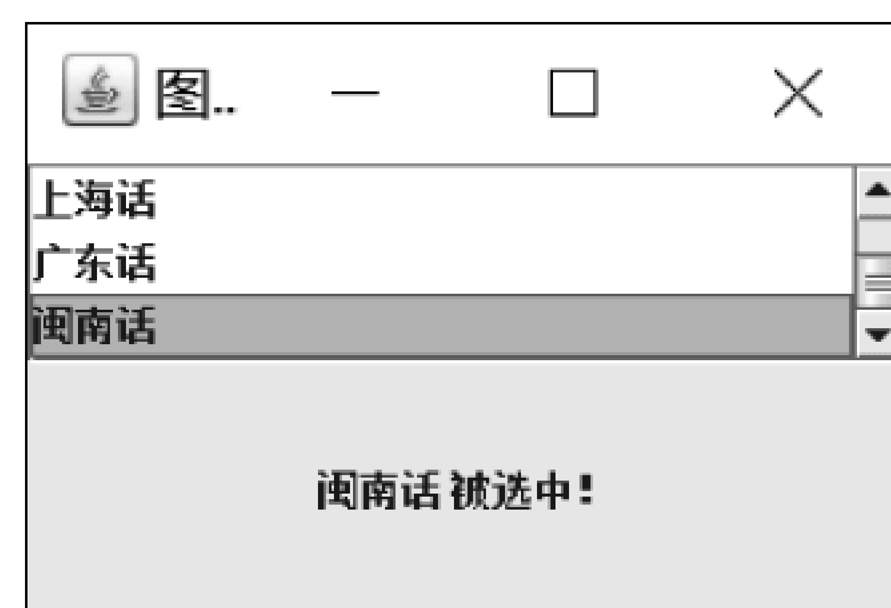


图 6-20 例 6-13 程序的运行结果

3. 表格类 JTable

使用列表类 JList<E>可以输出一维数组。如果想输出二维数组或二维表格,程序员可以使用表格类 **JTable**。例 6-14 给出一个表格类 JTable 的 Java 演示程序。

例 6-14 一个表格类 JTable 的 Java 演示程序(JTableTest.java)

```
 1  import java.awt.*;                    //导入 java.awt 包中定义的类
 2  //import java.awt.event.*;            //本例不需要导入 java.awt.event 包中的事件类
 3  import javax.swing.*;                 //导入 javax.swing 包中定义的类
 4
 5  public class JTableTest {                         //主类
 6      public static void main(String[] args) {      //主方法
 7          MainWnd w = new MainWnd();                //创建并显示主窗口对象
 8  } }
 9
10  class MainWnd extends JFrame {                    //扩展 JFrame
11      JTable list;                                  //二维表格
12      public MainWnd() {                            //构造方法
13          setTitle( "图形界面演示程序" );           //初始化窗口
14          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
```

```
15          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
16          //设置表格以及所要显示的数据
17          String columnNames[] = { "姓名", "成绩" }; //表格头(表格的列名)
18          Object data[][] = {  { "张三", 92 }, { "李四", 86 }, { "王五", 95 }  };  //数据
19          list = new JTable( data, columnNames);      //创建表格对象,初始化表格头和数据
20          //如果表格行数较多,需将表格放入一个卷滚面板
21          JScrollPane listScroller = new JScrollPane(list);
22          list.setFillsViewportHeight(true);          //设置是否填满显示区域
23          //在窗口的内容面板上添加组件
24          Container cp = getContentPane();            //获得窗口的内容面板(默认边框布局)
25          cp.add( listScroller, BorderLayout.CENTER );  //将表格卷滚面板添加到主窗口
26          cp.validate();                                //检查并自动布局容器里的组件
27  }  }
```

在 Eclipse 集成开发环境中运行例 6-14 的程序,运行结果如图 6-21 所示。

姓名	成绩
张三	92
李四	86
王五	95

图 6-21 例 6-14 程序的运行结果

6.4.6 菜单类

图形用户界面可以让用户选择程序功能。菜单是占用屏幕空间最少的一种形式。在框架窗口 JFrame 中添加菜单需分如下 4 步完成。

(1) 在框架窗口中添加一个菜单栏类 **JMenuBar** 的对象。

(2) 在菜单栏 JMenuBar 对象中添加一级菜单类 **JMenu** 的对象。

(3) 在一级菜单 JMenu 对象中添加二级菜单项类 **JMenuItem** 的对象。

(4) 为二级菜单项 JMenuItem 对象添加事件监听器,用于实现菜单所对应的程序功能。用户选择菜单会同时触发 **ItemEvent** 事件和 **ActionEvent** 事件。程序员需要为每个二级菜单项添加响应 ItemEvent 事件的 **ItemListener** 监听器,或添加响应 ActionEvent 事件的 **ActionListener** 监听器,二者任选其一。

例 6-15 给出一个为框架窗口添加菜单的 Java 演示程序。

例 6-15 一个为框架窗口添加菜单的 Java 演示程序(JMenuTest.java)

```
1~9   ......//第 1~9 行与 6.4.3 节例 6-8 相同,此处省略。注:将主类名改为 JMenuTest
 10   class MainWnd extends JFrame {                          //扩展 JFrame
 11       JMenuBar mb = new JMenuBar();                       //菜单栏
 12       JLabel info = new JLabel("信息显示区", SwingConstants.CENTER);  //信息标签
 13
 14       public MainWnd() {                                  //构造方法
 15           setTitle( "图形界面演示程序" );                    //初始化窗口
 16           setSize(300, 200);  setLocation(100, 100);  setVisible(true);
 17           setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 18           //处理 ActionEvent 事件的监听器:单击菜单则显示所选菜单项的名称
 19           ActionListener al = new ActionListener() {      //匿名类
 20               public void actionPerformed(ActionEvent e) {  //处理 ActionEvent 事件
```

```
21                JMenuItem src = (JMenuItem)(e.getSource());//获取事件源
22                String msg;
23                switch ( src.getText() ) {//显示所选择的菜单项信息
24                case "打开": msg = "选择菜单项:文件-打开!";  break;
25                case "保存": msg = "选择菜单项:文件-保存!";  break;
26                case "关闭": msg = "选择菜单项:文件-关闭!";  break;
27                case "复制": msg = "选择菜单项:编辑-复制!";  break;
28                case "剪切": msg = "选择菜单项:编辑-剪切!";  break;
29                case "粘贴": msg = "选择菜单项:编辑-粘贴!";  break;
30                default:  msg = src.getText();  break;
31                }
32                info.setText( msg );    //将信息显示在标签里
33            } };
34            //在框架窗口中添加菜单
35            setJMenuBar( mb );              //①在框架窗口中添加菜单栏对象 mb
36            JMenu m;   JMenuItem mi;        //定义局部引用变量 m 和 mi
37            //新建并添加一级菜单"文件"
38            m = new JMenu("文件");          //②新建一级菜单"文件"
39            mi = new JMenuItem("打开");     //③二级菜单项"打开"
40            mi.addActionListener( al );     //④为二级菜单项 mi 添加动作事件监听器 al
41            m.add( mi );                    //⑤将二级菜单项 mi 添加到一级菜单 m 中
42            mi = new JMenuItem("保存"); mi.addActionListener(al);  m.add(mi); //"保存"
43            mi = new JMenuItem("关闭"); mi.addActionListener(al);  m.add(mi); //"关闭"
44            mb.add( m );                    //⑥将一级菜单"文件"m 添加到菜单栏 mb 中
45            //新建并添加一级菜单"编辑"
46            m = new JMenu("编辑");          //新建一级菜单"编辑"
47            mi = new JMenuItem("复制"); mi.addActionListener(al);  m.add(mi); //"复制"
48            mi = new JMenuItem("剪切"); mi.addActionListener(al);  m.add(mi); //"剪切"
49            m.addSeparator();               //可在菜单中增加分隔线,用于功能分组
50            mi = new JMenuItem("粘贴"); mi.addActionListener(al);  m.add(mi); //"粘贴"
51            mb.add( m );                    //将一级菜单"编辑"m 添加到菜单栏 mb 中
52            //窗口及其内容面板的布局
53            Container cp = getContentPane();//获得窗口的内容面板(默认边框布局)
54            cp.add( info, BorderLayout.CENTER );    //添加信息显示区
55            cp.validate();                          //检查并自动布局内容面板里的组件
56            validate();                     //检查并自动布局窗口里的组件(包含菜单)
57    }  }
```

在 Eclipse 集成开发环境中运行例 6-15 的程序,运行结果如图 6-22 所示。

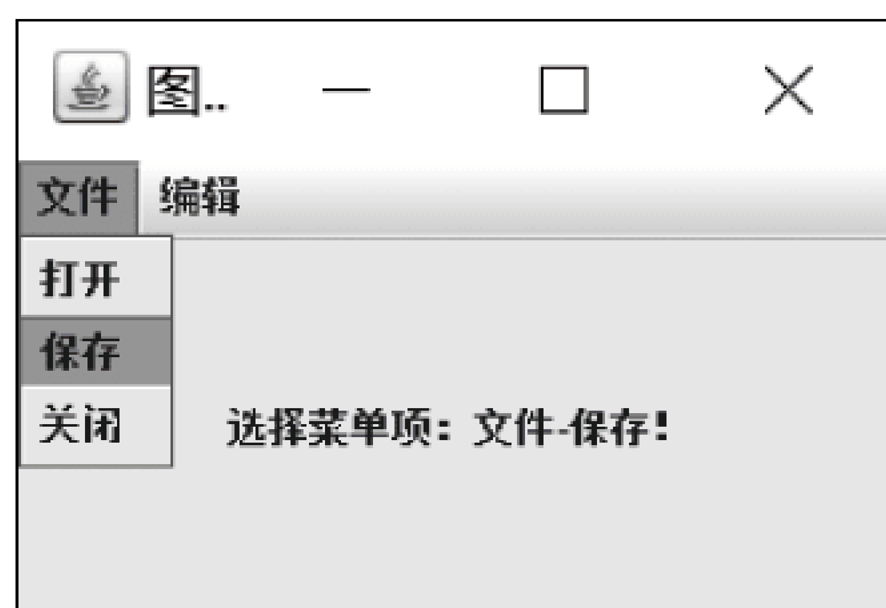


图 6-22　例 6-15 程序的运行结果

本节习题

1. 按钮类 JButton 中设置按钮名称的方法是(　　)。
   A. setText()　　B. getText()　　C. setTitle()　　D. drawString()
2. 标签类 JLabel 中显示图标(图像)的方法是(　　)。
   A. setIcon()　　B. getIcon()　　C. setTitle()　　D. drawImage()
3. 文本字段类 JTextField 对象通常需要响应(　　)事件。
   A. ActionEvent　　B. ItemEvent　　C. MouseEvent　　D. KeyEvent
4. 复选框类 JCheckBox 对象通常需要响应(　　)事件。
   A. ListSelectionEvent　　B. ItemEvent
   C. MouseEvent　　D. KeyEvent
5. 下拉列表类 JComboBox<E>对象通常需要响应(　　)事件。
   A. ListSelectionEvent　　B. ItemEvent
   C. MouseEvent　　D. KeyEvent
6. 列表类 JList<E>对象通常需要响应(　　)事件。
   A. ListSelectionEvent　　B. ItemEvent
   C. MouseEvent　　D. KeyEvent
7. 描述一级菜单项的类是(　　)。
   A. JMenuBar　　B. JMenu　　C. JMenuItem　　D. JTable
8. 二级菜单项类 JMenuItem 对象通常需要响应(　　)事件。
   A. ListSelectionEvent　　B. ActionEvent
   C. MouseEvent　　D. KeyEvent

6.5　对话框

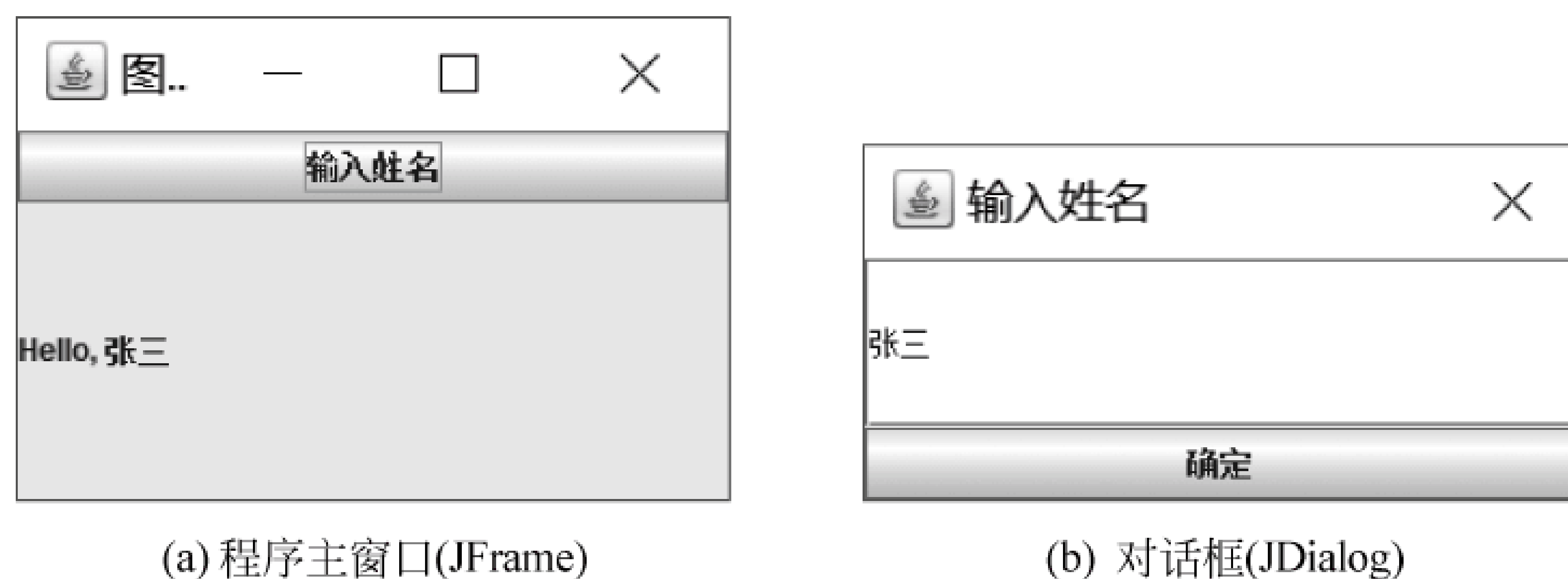
程序运行过程中,如果中途需要接收用户的指令或信息,然后再继续下一步运行,此时程序主窗口可以单独弹出一个**对话框**(dialog)来接收用户的输入。

对话框通常由框架窗口(JFrame,即程序主窗口)弹出,是隶属于框架窗口的子窗口。换句话说,框架窗口是对话框的父窗口。关闭框架窗口会同时关闭其所属的对话框。

和框架窗口一样,对话框也是顶级容器,可以包含其他组件,但对话框不能添加菜单栏。将程序中的部分界面功能独立出来,单独设计一个对话框,这样可以减轻程序主窗口的负担。

6.5.1　对话框类 JDialog

图 6-23 给出一个对话框演示程序,程序运行过程中需要用户输入一个姓名。用户单击图 6-23(a)程序主窗口中的“输入姓名”按钮,主窗口将单独弹出一个图 6-23(b)所示的对话框,用于接收用户输入的姓名。



(a) 程序主窗口(JFrame)　　(b) 对话框(JDialog)

图 6-23　程序主窗口弹出一个"输入姓名"的对话框

要实现图 6-23 的对话框功能，程序员需继承 Java API 中的对话框类 **JDialog**，然后在此基础上进行扩展，添加图形组件并响应事件。请读者阅读下面的对话框类 JDialog 说明文档。

javax. swing. **JDialog** 类说明文档

public class **JDialog**
extends **Dialog**
implements **WindowConstants**, **Accessible**, **RootPaneContainer**

	修饰符	类成员(节选)	功 能 说 明
1		**JDialog**()	构造方法
2		**JDialog**(Frame owner, String title)	构造方法
3		**JDialog**(Frame owner, String title, boolean modal)	构造方法
4		void **setTitle**(String title)	设置对话框标题
5		void **setModal**(boolean modal)	设置是否是模式对话框
6		Window **getOwner**()	获取父窗口
7		Container **getContentPane**()	获取内容面板
8		void **setLayout**(LayoutManager manager)	设置布局管理器
9	protected	void **dialogInit**()	初始化对话框的设置
10		void **setDefaultCloseOperation**(int operation)	设置关闭对话框时的默认操作
11		void **dispose**()	释放资源，关闭对话框

…

对话框分**模式**(modal，或称模态)和**非模式**(modaless，或称非模态)两种工作方式。模式对话框打开后，用户必须完成对话框所规定的数据输入或功能选择，然后才能返回父窗口，继续操作程序。简单地说，模式对话框打开后，本程序中的其他窗口都被禁止操作。而打开非模式对话框并不会影响用户操作本程序中的其他窗口，非模式对话框可以与其他窗口同时操作。模式对话框和非模式对话框在界面设计和代码实现上有一些区别，下面分别对它们进行讲解。

1. 模式对话框

如果程序功能要求用户必须在完成输入并关闭(或隐藏)对话框之后才能继续下一步操作，这时程序员应当将对话框设计成模式对话框。

例 6-16 给出一个使用模式对话框实现图 6-23 所示程序功能的 Java 示例程序。

例 6-16 使用模式对话框实现图 6-23 所示程序功能的 Java 示例程序(JDialogModalTest.java)

```
1~9  ......//第 1~9 行与 6.4.3 节例 6-8 相同,此处省略。注: 将主类名改为 JDialogModalTest
 10  class MainWnd extends JFrame {                  //主窗口类:扩展 JFrame
 11      JButton btn = new JButton("输入姓名"); //按钮:单击时弹出输入姓名对话框
 12      JLabel info = new JLabel("Hello!");     //信息标签
 13      JFrame fWin;                            //保存主窗口的引用,对话框需要访问该字段
 14      public MainWnd() {                      //构造方法
 15          setTitle( "图形界面演示程序" );     //初始化窗口
 16          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
 17          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 18          fWin = this;                        //保存当前窗口(主窗口)的引用
 19          //主窗口内容面板:按钮 + 信息标签
 20          Container cp = getContentPane();    //获得主窗口的内容面板(默认边框布局)
 21          cp.add( btn, BorderLayout.NORTH );  cp.add( info, BorderLayout.CENTER );
 22          cp.validate();                      //检查并自动布局主窗口内容面板里的组件
 23          btn.addActionListener( new ActionListener() {       //为按钮添加事件监听器
 24              public void actionPerformed( ActionEvent e) {
 25                  NameDialogM dlg = new NameDialogM( fWin); //创建对话框
 26                  dlg.setVisible(true);                     //弹出对话框
 27          }} );
 28  }  }
 29
 30  class NameDialogM extends JDialog {              //对话框类:扩展 JDialog
 31      JTextField name = new JTextField("张三"); //单行文本框:输入姓名
 32      JButton ok = new JButton("确定");          //按钮:确认输入
 33      JDialog dWin;                      //保存对话框的引用,事件监听器需要访问该字段
 34      public NameDialogM(JFrame pw) {          //构造方法,接收参数:父窗口
 35          super(pw);                           //调用超类 JDialog 的构造方法,传递父窗口
 36          setModal(true);                      //设置为模式对话框
 37          setTitle("输入姓名");  setSize(300, 150);  setLocation(200, 200);
 38          setDefaultCloseOperation( WindowConstants.HIDE_ON_CLOSE );
 39          dWin = this;                         //保存当前对话框的引用
 40          //对话框内容面板:单行文本框 + 按钮
 41          Container cp = getContentPane();  //获得对话框的内容面板(默认边框布局)
 42          cp.add( name, BorderLayout.CENTER );  cp.add( ok, BorderLayout.SOUTH );
 43          cp.validate();                       //检查并自动布局对话框内容面板里的组件
 44          //为单行编辑框和按钮添加事件监听器:接收输入,关闭对话框
 45          ActionListener al = new ActionListener() {          //匿名类
 46              public void actionPerformed(ActionEvent e) {     //处理 ActionEvent 事件
 47                  MainWnd w = ( MainWnd)dWin.getOwner();       //获取父窗口
 48                  w.info.setText("Hello, " + name.getText()); //读取并显示姓名
 49                  dWin.setVisible(false);                      //隐藏对话框
 50                  dWin.dispose();        //释放资源,关闭对话框
 51          } };
 52          name.addActionListener( al ); //输入姓名时按 Enter 键会触发 ActionEvent 事件
 53          ok.addActionListener( al );   //单击"确认"按钮会触发 ActionEvent 事件
 54  }  }
```

调用对话框方法成员 setModal()可以设置对话框的模式，例如例 6-16 中的代码第 36 行：

```
setModal(true);                //设置为模式对话框
```

模式对话框通常使用“确定”“取消”等按钮来结束输入，关闭对话框。程序员需为这些按钮添加事件监听器，当用户单击按钮时接收输入，释放资源并关闭对话框。

2. 非模式对话框

模式对话框打开后，本程序中的所有其他窗口都被禁止操作。而打开非模式对话框后，用户可以继续操作其他窗口，例如操作主窗口。非模式对话框在界面设计和代码实现上与模式对话框存在如下区别。

(1) 创建非模式对话框时，程序员需按如下方式来调用方法成员 setModal()：

```
setModal( false );             //设置为非模式对话框
```

(2) 程序应能够及时响应用户在非模式对话框中所做的输入，而不是必须单击“确定”“取消”等按钮才能接收输入。非模式对话框通常没有“确定”“取消”等按钮。

(3) 非模式对话框通常不需要频繁打开、关闭，而是由父窗口决定其是否可见(显示或隐藏)。

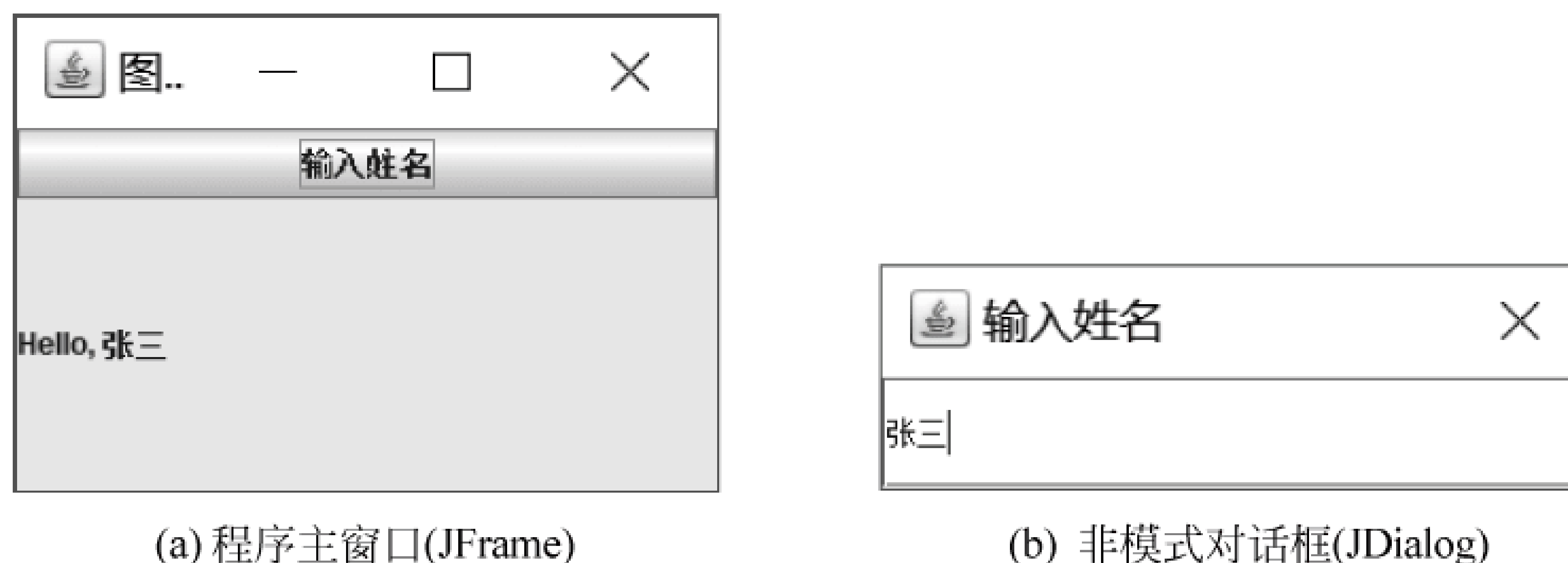
例 6-17 给出一个使用非模式对话框实现类似图 6-23 所示程序功能的 Java 示例程序。

例 6-17　使用非模式对话框实现类似图 6-23 所示程序功能的 Java 示例程序(JDialogModalessTest.java)

```
1～9  ......//第 1～9 行与 6.4.3 节例 6－8 相同,此处省略。注：将主类名改为 JDialogModalessTest
 10  class MainWnd extends JFrame {                        //主窗口类:扩展 JFrame
 11      JButton btn = new JButton("输入姓名");  //按钮:单击时弹出输入姓名对话框
 12      JLabel info = new JLabel("Hello!");     //信息标签
 13      JFrame fWin;              //保存主窗口的引用,对话框需要访问该字段
 14      NameDialogMless dlg;      //保存对话框的引用,事件监听器需要访问该字段
 15      public MainWnd() {                                //构造方法
 16          setTitle( "图形界面演示程序" );      //初始化窗口
 17          setSize(300, 200);  setLocation(100, 100);  setVisible(true);
 18          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
 19          fWin = this;                              //保存当前窗口(主窗口)的引用
 20          //主窗口内容面板:按钮 + 信息标签
 21          Container cp = getContentPane();     //获得主窗口的内容面板(默认边框布局)
 22          cp.add( btn, BorderLayout.NORTH );  cp.add( info, BorderLayout.CENTER );
 23          cp.validate();              //检查并自动布局主窗口内容面板里的组件
 24          dlg = new NameDialogMless( fWin );   //创建输入姓名的非模式对话框
 25          btn.addActionListener( new ActionListener() {      //为按钮添加事件监听器
 26              public void actionPerformed( ActionEvent e) {
 27                  dlg.setVisible(true);     //显示非模式对话框(不是每次都重新创建)
 28          }} );
 29  }  }
 30
 31  class NameDialogMless extends JDialog {               //对话框类:扩展 JDialog
 32      JTextField name = new JTextField("张三");     //单行文本框:输入姓名
 33      JDialog dWin;                  //保存对话框的引用,事件监听器需要访问该字段
 34      public NameDialogMless(JFrame pw) {                //构造方法
```

```
35        super(pw);              //调用超类 JDialog 的构造方法,传递父窗口
36        setModal(false);        //设置为非模式对话框
37        setTitle("输入姓名");  setSize(300, 100);  setLocation(400, 200);
38        setDefaultCloseOperation( WindowConstants.HIDE_ON_CLOSE );
39        dWin = this;            //保存当前对话框的引用
40        //对话框内容面板:只有一个单行文本框
41        Container cp = getContentPane();    //获得对话框的内容面板(默认边框布局)
42        cp.add( name, BorderLayout.CENTER );cp.validate();
43        //为单行编辑框添加事件监听器:接收输入,输入后不需要关闭对话框
44        ActionListener al = new ActionListener() {          //匿名类
45            public void actionPerformed( ActionEvent e) {
46                MainWnd w = (MainWnd)dWin.getOwner();       //获取父窗口
47                w.info.setText("Hello, " + name.getText());  //读取并显示姓名
48                //非模式对话框不是必须关闭,可与程序主窗口并存
49        } };
50        name.addActionListener( al );  //输入姓名时按 Enter 键会触发 ActionEvent 事件
51  }  }
```

在 Eclipse 集成开发环境中运行例 6-17 的程序,运行结果如图 6-24 所示。用户单击图 6-24(a)程序主窗口中的“输入姓名”按钮,主窗口将单独弹出一个图 6-24(b)所示的非模式对话框。在这个非模式对话框的单行文本框中输入姓名并按 Enter 键,可以立即在主窗口的信息标签中看到所输入的姓名。程序运行过程中,非模式对话框可与程序主窗口并存。



(a) 程序主窗口(JFrame)　　(b) 非模式对话框(JDialog)

图 6-24　程序主窗口与非模式对话框并存

6.5.2　常用对话框

图形用户界面程序有一些常用的对话框,例如**消息**(message)对话框、**确认**(confirm)对话框、**输入**(input)对话框、**选项**(option)对话框等。Java API 预先设计好了这些对话框功能,并以静态方法的形式提供给程序员使用。这些静态方法被统一定义在选项面板类 **JOptionPane** 中。程序员只要调用选项面板类 JOptionPane 中的静态方法,就能很方便地实现上述常用的对话框功能。

另外还有两个比较常用的对话框,即**文件选择**对话框和**颜色选择**对话框。因为这两个对话框比较复杂,因此 Java API 为它们单独定义了类,这就是文件选择类 **JFileChooser** 和颜色选择类 **JColorChooser**。

1. 选项面板类 JOptionPane

请读者阅读下面的选项面板类 JOptionPane 说明文档。

javax.swing.**JOptionPane** 类说明文档

public class **JOptionPane**
extends **JComponent**
implements **Accessible**

	修饰符	类成员(节选)	功能说明
1	static	void **showMessageDialog**(Component parentComponent, Object message)	消息对话框
2	static	void **showMessageDialog**(Component parentComponent, Object message, String title, int messageType)	消息对话框
3	static	int **showConfirmDialog**(Component parentComponent, Object message)	确认对话框
4	static	int **showConfirmDialog**(Component parentComponent, Object message, String title, int optionType)	确认对话框
5	static	String **showInputDialog**(Component parentComponent, Object message)	输入对话框
6	static	String **showInputDialog**(Component parentComponent, Object message, Object initialSelectionValue)	输入对话框
7	static	int **showOptionDialog**(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)	选项对话框

…

1) 消息对话框

消息对话框主要用于向用户显示提示信息。例 6-18 给出一个弹出消息对话框的 Java 演示程序。

例 6-18 一个弹出消息对话框的 Java 演示程序(JOptionPaneTest.java)

```
 1  import java.awt.*;                            //导入 java.awt 包中的类
 2  import java.awt.event.*;                      //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                         //导入 javax.swing 包中的类
 4
 5  public class JOptionPaneTest {                //测试类
 6      public static void main(String[] args) {  //主方法
 7          JFrame w = new JFrame();              //创建并显示主窗口对象
 8          w.setTitle( "图形界面演示程序" );       //初始化主窗口
 9          w.setSize(300, 200);  w.setLocation(100, 100);  w.setVisible(true);
10          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11          //主窗口内容面板:按钮 + 标签
12          Container cp = w.getContentPane();    //获得窗口的内容面板(默认边框布局)
13          JButton btn = new JButton("测试对话框");         //按钮:单击按钮弹出对话框
14          JLabel info = new JLabel("Hello!");           //标签:显示对话框返回的结果
15          cp.add( btn, BorderLayout.NORTH );  cp.add( info, BorderLayout.CENTER );
```

```
16          cp.validate();                          //检查并自动布局内容面板里的组件
17          //为按钮添加事件监听器:用户单击按钮,弹出对话框
18          btn.addActionListener( new ActionListener() {          //匿名类
19              public void actionPerformed(ActionEvent e) {       //弹出消息对话框
20                  JOptionPane.showMessageDialog(w, "欢迎进入课程学习",
21                      Java 语言程序设计, JOptionPane.INFORMATION_MESSAGE);
22          }} );
23  }  }
```

在 Eclipse 集成开发环境中运行例 6-18 的程序,运行结果如图 6-25 所示。用户单击图 6-25(a)程序主窗口中的"测试对话框"按钮,主窗口将弹出一个图 6-25(b)所示的消息对话框。这个消息对话框显示了一个课程欢迎信息,用户单击"确定"按钮即可关闭对话框。**注**:消息对话框不返回任何结果。



(a) 程序主窗口



(b) 消息对话框

图 6-25　例 6-18 程序的主窗口及其弹出的消息对话框

2) 确认对话框

当需要用户确认某个程序功能或选项时,程序员可使用确认对话框。按如下形式修改例 6-18 中的代码第 20～21 行,演示程序将弹出确认对话框,如图 6-26 所示。

```
int opt;                                  //接收确认对话框返回的用户选择(int 类型)
opt = JOptionPane.showConfirmDialog(w, "进入课程学习吗?",  //弹出确认对话框
    "Java 语言程序设计", JOptionPane.INFORMATION_MESSAGE);
if (opt == JOptionPane.YES_OPTION)        //根据返回结果 opt 显示用户的确认结果
    info.setText("您选择了是");
else if (opt == JOptionPane.NO_OPTION)
    info.setText("您选择了否");
else if (opt == JOptionPane.CANCEL_OPTION)
    info.setText("您选择了取消");
```

3) 输入对话框

当需要用户输入某个参数时,程序员可使用输入对话框。按如下形式修改例 6-18 中的代码第 20～21 行,演示程序将弹出输入对话框,如图 6-27 所示。

```
String str;                                  //接收输入对话框返回的用户输入(字符串类型)
str = JOptionPane.showInputDialog(w, "请输入姓名:");    //弹出输入对话框
info.setText("您输入的是 " + str);           //根据返回结果 str 显示用户的输入
```

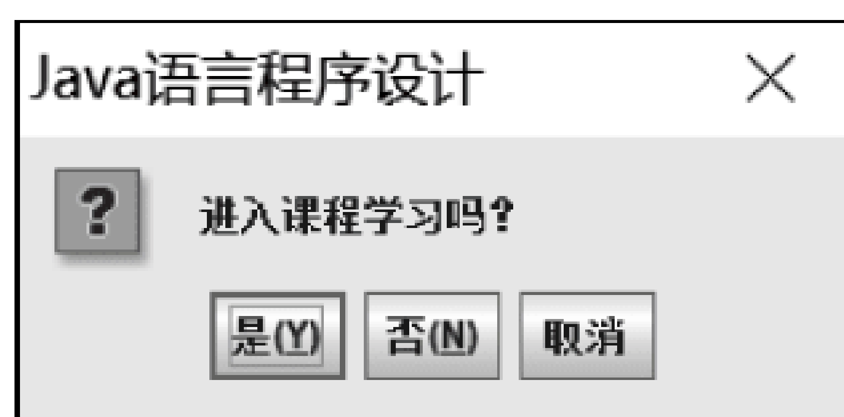


图 6-26　确认对话框

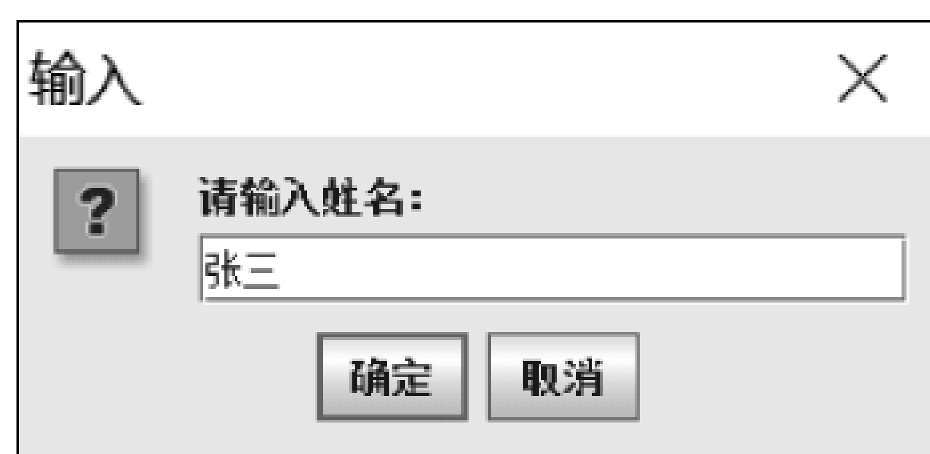


图 6-27　“输入”对话框

4）选项对话框

当需要用户在多个选项中进行选择时，程序员可使用选项对话框。按如下形式修改例 6-18 中的代码第 20～21 行，演示程序将弹出选项对话框，如图 6-28 所示。

```
String gender[] = { "男", "女" };             //提交用户选择的选项数组(通常为字符串类型)
int opt = JOptionPane.showOptionDialog(       //弹出选项对话框
    w, "请选择性别", "Java 语言程序设计",
    JOptionPane.DEFAULT_OPTION, JOptionPane.PLAIN_MESSAGE,null, gender, "男");
info.setText("您选择的是 " + gender[opt]);     //根据返回值 opt 显示用户的选择结果
```

2. 文件选择类 JFileChooser

打开或保存文件需要用户输入或选择文件名，这时程序员可以直接使用 Java API 提供的文件选择类 JFileChooser。使用文件选择类 JFileChooser 可以很方便地创建文件选择对话框，接收用户输入或选择的文件名。

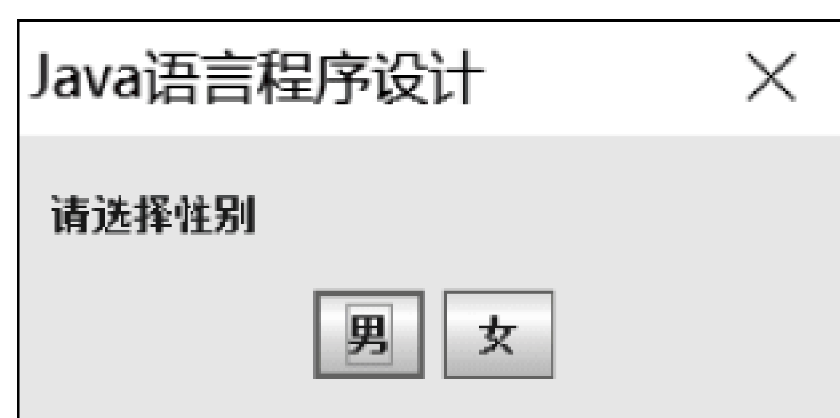


图 6-28　选项对话框

请读者阅读下面的文件选择类 JFileChooser 说明文档。

javax.swing.**JFileChooser** 类说明文档

public class **JFileChooser**
extends **JComponent**
implements **Accessible**

	修饰符	类成员(节选)	功 能 说 明
1		**JFileChooser**()	构造方法
2		**JFileChooser**(String currentDirectoryPath)	构造方法
3		void **setCurrentDirectory**(File dir)	设置当前目录
4		void **setFileFilter**(FileFilter filter)	设置文件过滤器，例如只显示.jpg 图像文件
5		void **setMultiSelectionEnabled**(boolean b)	设置是否可以多选
6		int **showOpenDialog**(Component parent)	弹出一个打开文件对话框
7		int **showSaveDialog**(Component parent)	弹出一个保存文件对话框
8		int **showDialog**(Component parent,String approveButtonText)	弹出一个文件选择对话框
9		File **getSelectedFile**()	获取所选择的文件
10		File[] **getSelectedFiles**()	获取所选择的文件列表
11		String **getName**(File f)	获取文件 f 的文件名

…

注：文件选择类 JFileChooser 说明文档中的文件类 File 将在第 7 章讲解。

例 6-19 给出一个文件选择类 JFileChooser 的 Java 演示程序。

例 6-19 一个文件选择类 JFileChooser 的 Java 演示程序(JFileChooserTest.java)

```
1  import javax.swing.*;                             //导入 javax.swing 包中的类
2  import java.io.File;                              //导入 java.io 包中定义的文件类 File
3
4  public class JFileChooserTest {                   //测试类
5      public static void main(String[] args) {      //主方法
6          JFileChooser fc = new JFileChooser();     //创建一个文件选择类的对象
7          fc.setCurrentDirectory( new File("D:/我的 Java 语言/Example/Chapter1/src") );
8          fc.showOpenDialog(null);                  //弹出打开文件对话框
9          File f = fc.getSelectedFile();            //返回所选择的文件
10         JOptionPane.showMessageDialog(null, "您选择的文件是:" + f.getName());
11 } }
```

在 Eclipse 集成开发环境中运行例 6-19 的程序,运行结果如图 6-29 所示。用户可以在图 6-29 中选择某个希望打开的文件。文件选择类 JFileChooser 的对象还可以作为图形组件,被放入到某个容器中。

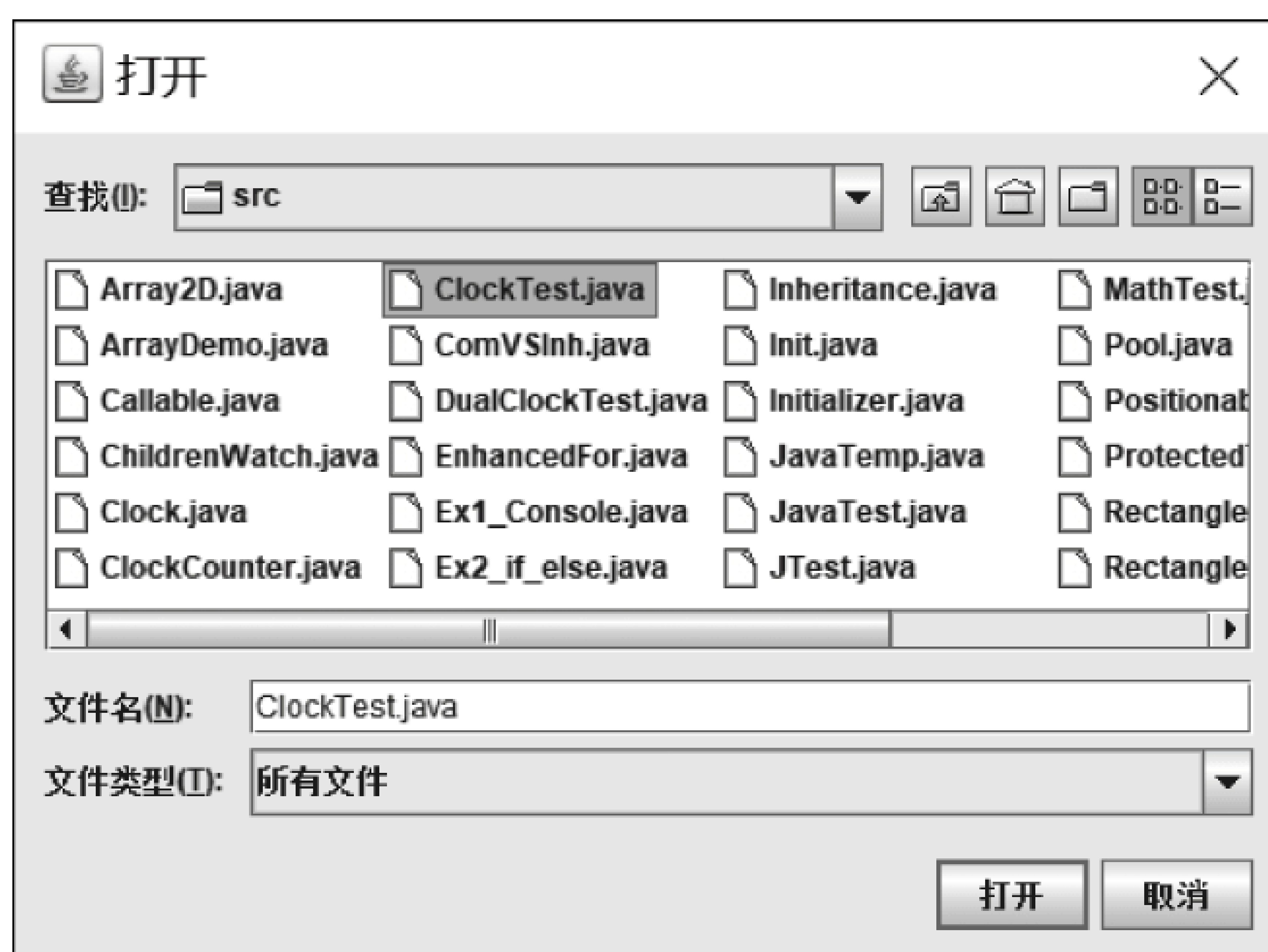


图 6-29 文件选择-打开文件对话框

3. 颜色选择类 JColorChooser

绘图时可能需要用户选择颜色,程序员可以直接使用 Java API 提供的颜色选择类 JColorChooser。使用颜色选择类 JColorChooser 可以很方便地创建颜色选择对话框,接收用户输入或选择的颜色。

请读者阅读下面的颜色选择类 JColorChooser 说明文档。

javax. swing. **JColorChooser** 类说明文档			
public class **JColorChooser** extends **JComponent** implements **Accessible**			
	修饰符	类成员(节选)	功 能 说 明
1		**JColorChooser**()	构造方法
2		**JColorChooser**(Color initialColor)	构造方法
3		void **setColor**(Color color)	设置当前颜色
4		Color **getColor**()	获取当前颜色
5	static	Color **showDialog** (Component component, String title, Color initialColor)	显示对话框,选择颜色
…			

例 6-20 给出一个颜色选择类 JColorChooser 的 Java 演示程序。

例 6-20　一个颜色选择类 JColorChooser 的 Java 演示程序(JColorChooserTest. java)

```
1  import javax.swing.*;                 //导入 javax.swing 包中的类
2  import java.awt.Color;                //导入 java.awt 包中定义的颜色类 Color
3
4  public class JColorChooserTest {                        //测试类
5      public static void main(String[] args) {            //主方法
6          JColorChooser cc = new JColorChooser();         //创建一个颜色选择类的对象
7          Color c = cc.showDialog(null, "请选择颜色", Color.RED); //弹出选择颜色对话框
8          JOptionPane.showMessageDialog(null, "您选择的颜色是:" + c.toString());
9  } }
```

在 Eclipse 集成开发环境中运行例 6-20 的程序,运行结果如图 6-30 所示。用户可以在图 6-30 中选择某种需要的颜色。颜色选择类 JColorChooser 的对象还可以作为图形组件,被放入到某个容器中。

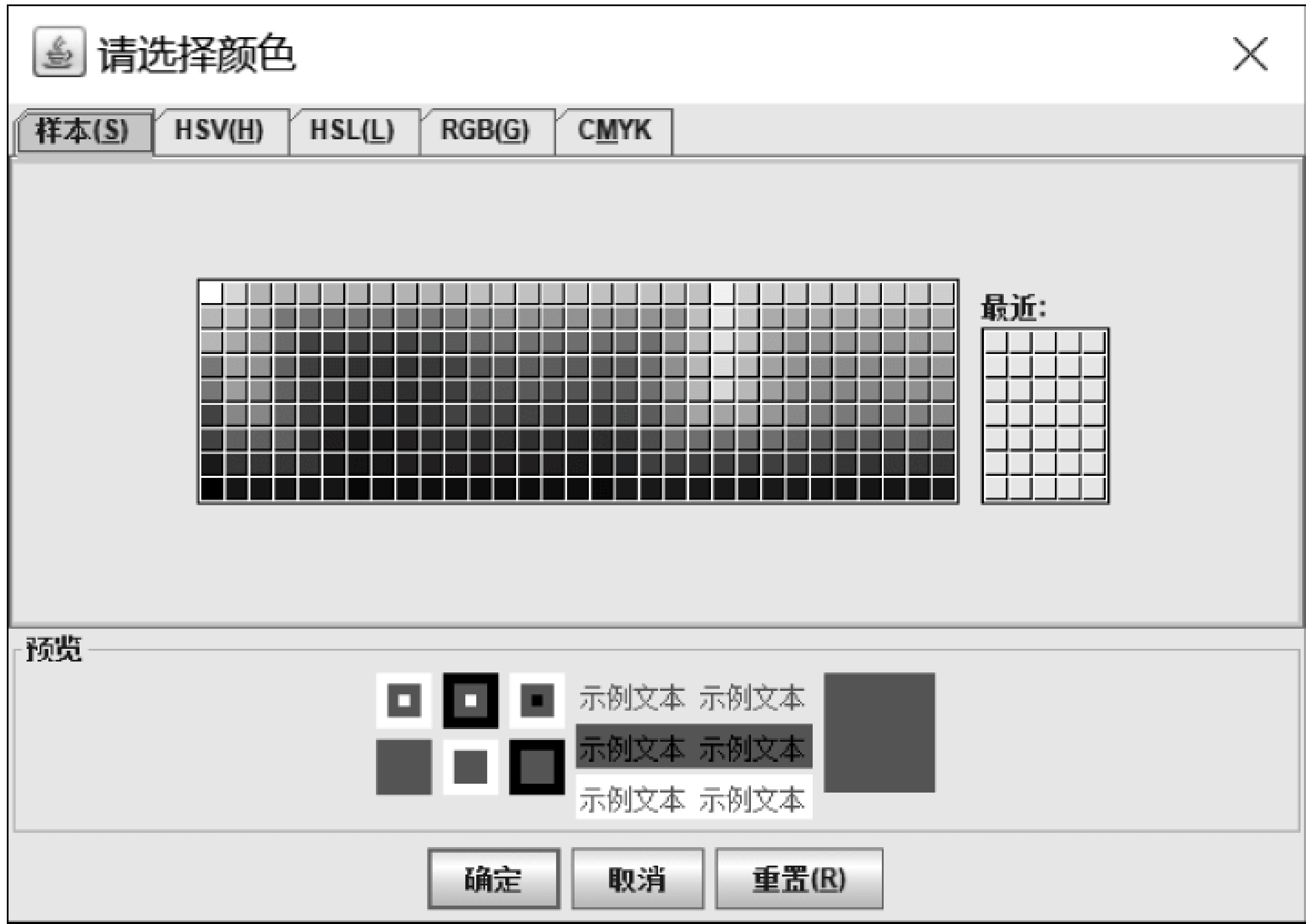


图 6-30　颜色选择对话框

本节习题

1. 下列关于对话框的描述中,错误的是(　　)。
   A. 对话框是框架窗口单独弹出的一个子窗口
   B. 对话框可以添加菜单栏
   C. 对话框类 JDialog 是一个顶层容器
   D. 对话框分模式和非模式两种
2. 对话框类 JDialog 中设置是否是模式对话框的方法是(　　)。
   A. setModal()　　　　B. setLayout()
   C. setTitle()　　　　D. setDefaultCloseOperation()
3. 对话框类 JDialog 中取得内容面板的方法是(　　)。
   A. getContentPane()　　　　B. getOwner()
   C. getGraphics()　　　　D. getWidth()
4. 没有在选项面板类 JOptionPane 中提供的对话框是(　　)。
   A. 消息(message)对话框　　　　B. 确认(confirm)对话框
   C. 输入(input)对话框　　　　D. 文件选择(JFileChooser)对话框
5. 选项面板类 JOptionPane 中弹出选项对话框的方法是(　　)。
   A. showMessageDialog()　　　　B. showConfirmDialog()
   C. showInputDialog()　　　　D. showOptionDialog()

6.6 鼠标事件和键盘事件

图形用户界面中的所有图形组件都有**鼠标事件**(MouseEvent)和**键盘事件**(KeyEvent),这是两个**底层事件**(low-level event)。为了方便程序员,Java API 将底层事件提炼成适合编程使用的高层**语义事件**(semantic event),例如之前介绍过的 ActionEvent、ItemEvent、ListSelectionEvent 等。通常,程序员不需要处理鼠标或键盘事件。

程序员也可以为图形组件添加监听器来处理底层的鼠标和键盘事件,其代表性应用场合是计算机绘图程序。鼠标和键盘事件,以及它们对应的监听器接口,请参见 6.3.3 节。

6.6.1 响应鼠标和键盘事件

Java API 为鼠标事件设计了两个监听器接口:一是鼠标监听器接口 **MouseListener**,可以处理鼠标进出组件或鼠标按键操作事件;另一个是鼠标移动监听器接口 **MouseMotionListener**,可以处理鼠标移动或拖动(按下按键的同时移动鼠标)事件。Java API 为键盘事件设计了键盘监听器接口 **KeyListener**,可以处理键盘按键操作事件。

如果需要处理图形组件的鼠标和键盘事件,程序员需为组件添加相应的事件监听器。例 6-21 给出了一个响应底层鼠标和键盘事件的 Java 演示程序。

例 6-21 一个响应底层鼠标和键盘事件的 Java 演示程序(JMouseKeyTest.java)

```
1  import java.awt.*;                                //导入 java.awt 包中的类
2  import java.awt.event.*;                          //导入 java.awt.event 包中定义的事件类
3  import javax.swing.*;                             //导入 javax.swing 包中的类
4
5  public class JMouseKeyTest {                      //测试类
6      public static void main(String[] args) {      //主方法
7          JFrame w = new JFrame();                  //创建并显示主窗口对象
8          w.setTitle( "图形界面演示程序" );          //初始化窗口
9          w.setSize(300, 200);  w.setLocation(100, 100);  w.setVisible(true);
10         w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11         //主窗口内容面板:添加一个显示信息的标签
12         Container cp = w.getContentPane();  //获得主窗口的内容面板(默认边框布局)
13         JLabel info = new JLabel("检测键盘和鼠标动作");
14         cp.add(info, BorderLayout.SOUTH);         //添加信息标签
15         cp.setBackground(Color.YELLOW);           //设置内容面板的背景色
16         cp.validate();                            //检查并自动布局容器里的组件
17         //为内容面板容器添加鼠标事件监听器对象
18         w.addMouseListener( new MouseListener() {             //匿名类
19             public void mouseClicked(MouseEvent e)            //单击了鼠标按键
20             {  info.setText("单击了鼠标按键");  }
21             public void mouseEntered(MouseEvent e)            //鼠标进入内容面板区域
22             {  info.setText("鼠标进入内容面板区域");  }
23             public void mouseExited(MouseEvent e)             //鼠标离开内容面板区域
24             {  info.setText("鼠标离开内容面板区域");  }
25             public void mousePressed(MouseEvent e) { }        //鼠标按键被按下,未响应
26             public void mouseReleased(MouseEvent e) { }       //鼠标按键被松开,未响应
27         } );
28         //为内容面板容器添加鼠标移动事件监听器
29         cp.addMouseMotionListener( new MouseMotionListener() {     //匿名类
30             public void mouseMoved(MouseEvent e) {            //鼠标被移动
31                 String str = String.format( "鼠标在移动: %d, %d", e.getX(), e.getY() );
32                 info.setText(str);                            //显示当前鼠标的位置坐标
33             }
34             public void mouseDragged(MouseEvent e) {          //鼠标被拖动
35                 String str = String.format( "鼠标在拖动: %d, %d", e.getX(), e.getY() );
36                 info.setText(str);                            //显示当前鼠标的位置坐标
37         }} );
38         //为内容面板容器添加键盘事件监听器
39         cp.requestFocus();                                    //获得键盘输入焦点
40         cp.addKeyListener( new KeyListener() {                //匿名类
41             public void keyTyped(KeyEvent e) {                //敲击了某个键盘的按键
42                 String str = String.format( "敲击了按键: %c", e.getKeyChar() );
43                 info.setText(str);                            //显示被敲击的按键
44             }
45             public void keyPressed(KeyEvent e) { }            //某个按键被按下,未响应
46             public void keyReleased(KeyEvent e) { }           //某个按键被松开,未响应
47         } );
48 }  }
```

需要注意的是,处理鼠标和键盘事件的监听器接口中都定义了多个抽象方法。在编写响应鼠标和键盘事件的监听器代码时,即使只响应部分操作也需要实现监听器接口中的所有抽象方法,哪怕是定义一个空的方法。例如,例 6-21 中代码第 18~27 行在编写响应鼠标事件 MouseEvent 的监听器代码时,没有响应按下或松开鼠标按键的操作,但仍需要实现 mousePressed()和 mouseReleased()这两个抽象方法,将它们定义成空方法。



图 6-31 响应内容面板区域的鼠标或键盘事件

在 Eclipse 集成开发环境中运行例 6-21 的程序,运行结果如图 6-31 所示。在图 6-31 中窗口的内容面板区域操作鼠标或敲击键盘按键,程序将立即捕捉到鼠标或键盘事件,然后在窗口下方的信息标签中显示出相关信息。

6.6.2 在画布上绘图

Java API 提供了一个专门用于绘图的画布类 **Canvas**。它是一个图形组件,描述了一个可以绘图的矩形区域。使用画布类 Canvas 编写绘图程序的基本步骤如下。

(1) 继承并扩展画布类 Canvas,重写其中的绘图方法 **paint**()。

(2) 使用扩展后的画布类创建画布对象。

(3) 为画布对象添加响应鼠标和键盘事件的监听器,记录用户在画布上的操作。

(4) 每次记录完用户操作之后即调用画布类的重画方法 **repaint**(),对画布进行刷新、重绘。

例 6-22 给出一个使用画布类 Canvas 编写的绘图演示程序。

例 6-22 一个使用画布类 Canvas 编写的绘图演示程序(JCanvasTest.java)

```
 1  import java.awt.*;                          //导入 java.awt 包中的类
 2  import java.awt.event.*;                    //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                       //导入 javax.swing 包中的类
 4
 5  public class JCanvasTest {                  //测试类
 6      public static void main(String[] args) { //主方法
 7          JFrame w = new JFrame();            //创建并显示主窗口对象
 8          w.setTitle( "图形界面演示程序" );   //初始化主窗口
 9          w.setSize(400, 300);  w.setLocation(100, 100);  w.setVisible(true);
10          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11          //主窗口内容面板:标签 + 画布
12          Container cp = w.getContentPane(); //获得主窗口的内容面板(默认边框布局)
13          JLabel info = new JLabel("在画布上单击鼠标显示字符,或敲击键盘按键选择字符");
14          MyCanvas cv = new MyCanvas();       //创建用于绘图的画布对象
15          cp.add(info, BorderLayout.NORTH);   //添加信息标签
16          cp.add(cv, BorderLayout.CENTER);    //添加绘图画布
17          cp.validate();                      //检查并自动布局容器里的组件
18          //用户单击画布,在鼠标光标位置画一个圆,然后在圆中显示一个字符
19          //为画布添加鼠标事件监听器,将单击鼠标的位置坐标记录在画布对象中
20          cv.addMouseListener( new MouseListener() {          //匿名类
```

```
21              public void mouseClicked(MouseEvent e) {
22                  cv.x = e.getX();   cv.y = e.getY();     //记录鼠标位置
23                  cv.repaint();                           //对画布进行重绘刷新
24              }
25              public void mouseEntered(MouseEvent e) { }  //以下 4 个事件不响应
26              public void mouseExited(MouseEvent e) { }
27              public void mousePressed(MouseEvent e) { }
28              public void mouseReleased(MouseEvent e) { }
29          } );
30          //为画布添加键盘事件监听器,将所敲击的按键记录在画布对象中
31          cv.addKeyListener( new KeyListener() {          //匿名类
32              public void keyTyped(KeyEvent e) {
33                  cv.key = e.getKeyChar();                //记录键盘按键
34              }
35              public void keyPressed(KeyEvent e) { }      //以下 2 个事件不响应
36              public void keyReleased(KeyEvent e) { }
37          } );
38  }  }
39
40  class MyCanvas extends Canvas {                     //定义自己的画布类,继承 Canvas 类
41      public int x = -1, y = -1;                      //添加记录鼠标位置的字段
42      public char key = '+';                          //添加记录键盘按键的字段,初始值为"+"
43      private Font ef = new Font("TimesRoman", Font.PLAIN, 32);  //字体
44      public MyCanvas() {                             //构造方法
45          setBackground(Color.YELLOW);                //设置画布背景色
46      }
47      public void paint(Graphics g) {                 //重写 paint()方法
48          if (x == -1 || y == -1)  return;            //未单击过鼠标,直接返回
49          g.drawOval(x-5, y-25, 27, 27);              //在单击鼠标的位置画一个圆
50          g.setFont( ef );                            //设置字体
51          g.drawString( String.valueOf(key), x, y );     //在圆中显示一个字符
52  }  }
```

在 Eclipse 集成开发环境中运行例 6-22 的程序,运行结果如图 6-32 所示。在图 6-32 中窗口的画布区域操作鼠标或敲击键盘按键,程序将立即捕捉到鼠标或键盘事件,然后在鼠标单击位置进行绘图。



图 6-32　响应画布的鼠标或键盘事件进行绘图

本节习题

1. 下列图形用户界面事件中,属于底层事件的是(   )。

A. MouseEvent　　B. ItemEven
C. ActionEvent　　D. ListSelectionEvent

2. 下列方法中,没有定义在处理鼠标事件 MouseEvent 的监听器接口 MouseListener 中的方法是(   )。

A. mouseClicked()　　B. mouseEntered()
C. mouseReleased()　　D. mouseMoved()

3. 下列方法中,定义在处理鼠标事件 MouseEvent 的监听器接口 MouseMotionListener 中的方法是(   )。

A. mouseClicked()　　B. mouseEntered()
C. mouseReleased()　　D. mouseDragged()

4. 下列方法中,没有定义在处理键盘事件 KeyEvent 的监听器接口 KeyListener 中的方法是(   )。

A. keyTyped()　　B. keyPressed()
C. keyReleased()　　D. keyDown()

5. 当需要在组件上绘图时,程序通常应当重写组件类的(   )方法。

A. repaint()　　B. paint()
C. update()　　D. validate()

6.7 Java 小应用程序类 Applet

Java 小应用程序(applet)是一种特殊的图形用户界面程序。编写 Java 小应用程序需要继承 java.applet 包中的小应用程序类 **Applet**。这个类是 Java 抽象窗口工具包 awt 提供的一种图形组件,程序可以在该图形组件上显示文字或绘制图形、图像等。

1. Java 小应用程序示例

例 6-23 给出一个简单的 Java 小应用程序示例代码。

例 6-23 一个简单的 Java 小应用程序示例代码(AppletDemo.java)

```
1  import java.applet.*;                    //导入 java.applet 包中与小应用程序相关的类和接口
2  import java.awt.*;                       //导入 java.awt 包中的类
3
4  public class AppletDemo extends Applet {            //定义一个小应用程序类
5      String msg;                                      //显示用的文字信息
6      Font f;                                          //显示用的字体
7      Color c;                                         //绘图用的颜色
8      public void init() {                             //重写超类 Applet 的初始化方法
```

```
 9             msg = "Hello, World";                       //将在组件上显示该字符串
10             f = new Font("TimesRoman", Font.PLAIN, 16);//字体
11             c = Color.RED;                              //颜色
12         }
13         public void paint(Graphics g) {                 //重写绘制方法:显示文字、绘制图形
14             super.paint( g );                           //调用超类的 paint()方法
15             g.setFont( f );                             //设置字体
16             g.drawString(msg, 20, 40);                  //显示文字信息"Hello, World"
17             g.setColor( c );                            //设置绘图颜色
18             g.fillRect(20, 60, 100, 100);               //填充一个实心的正方形
19         }
20     }
```

Java 小应用程序不需要定义主方法 main()，可以直接在 Eclipse 中运行（**Run As** Java Applet）。在 Eclipse 中运行例 6-23 的 Java 小应用程序，其运行结果如图 6-33 所示。

2. 在浏览器中运行 Java 小应用程序



图 6-33　例 6-23 中 Java 小应用程序的运行结果

Java 语言提供小应用程序的目的是在浏览器（browser）中运行，用于增强 HTML 网页的表现力。例如，在 HTML 网页中嵌入 Java 小应用程序可以显示文字、绘制图形或图像、播放音频或动画，或为用户提供 Java 风格的图形用户界面等。

在 HTML 网页中嵌入 Java 小应用程序的语法形式如下：

```
<applet code="小应用程序文件名" height=高度 width=宽度>
</applet>
```

例如，可以按如下形式将例 6-23 的 Java 小应用程序 AppletDemo.class 嵌入到 HTML 网页中。

```
<html>
    <applet code= "AppletDemo.class"   height=300 width=200>
    </applet>
</html>
```

使用浏览器打开这个 HTML 网页，将会在浏览器窗口显示出与图 6-33 完全相同的内容。也可以使用 JDK 提供的小应用程序查看器（\JDK 安装目录\bin**appletviewer**.exe）来查看 HTML 网页中的小应用程序。例如，在 Windows 控制台状态下查看上述 HTML 网页的命令为：

appletviewer　HTML 网页文件名 <Enter 键>

通常，Java 小应用程序在开发过程中是通过集成开发环境或小应用程序查看器来运行、调试的，而实际交付使用后则是在浏览器中运行的。

3. 小应用程序类 Applet

小应用程序类 Applet 是 awt 中面板类 **Panel** 的子类，因此它是一个图形容器，而且还

是一个顶层容器。可以在小应用程序类 Applet 容器中添加各种其他图形组件。请读者阅读下面的小应用程序类 Applet 说明文档。

java. applet. **Applet** 类说明文档

public class **Applet**
extends **Panel**

	修饰符	类成员(节选)	功 能 说 明
1		**Applet**()	构造方法
2		void **init**()	浏览器加载 Applet 后自动调用该方法,用于定义 Applet 的初始化代码
3		void **start**()	浏览器在调用完 init()方法后自动调用该方法,用于定义 Applet 的功能代码
4		void **stop**()	在退出 Applet 所在页面时浏览器自动调用该方法,用于定义 Applet 的暂停代码
5		void **destroy**()	浏览器在删除 Applet 时自动调用该方法,用于定义 Applet 的善后处理代码
6		Image **getImage**(URL url)	加载 url 指定的图像文件,用于后续显示
7		AudioClip **getAudioClip**(URL url)	加载 url 指定的音频文件,用于后续播放
8		void **play**(URL url)	直接播放 url 指定的音频文件

…

小应用程序类 Applet 继承面板类 Panel,另外还扩展了几个与程序运行相关的方法,例如 init()、start()、stop()、destroy()等。小应用程序类 Applet 还定义了两个与图像、音频相关的方法。

加载图像文件方法 **getImage**()可以将图像文件加载到内存,生成一个图像类 **Image** 的对象。后续程序可以显示这个图像对象。请读者阅读下面的图像类 Image 说明文档。

java. awt. **Image** 类说明文档

public abstract class **Image**
extends Object

	修饰符	类成员(节选)	功 能 说 明
1		**Image**()	构造方法
2	abstract	int **getWidth**(ImageObserver observer)	获取图像宽度
3	abstract	int **getHeight**(ImageObserver observer)	获取图像高度

…

加载音频文件方法 **getAudioClip**()可以将音频文件加载到内存,生成一个音频片段接口 **AudioClip** 的对象。后续程序可以播放这个音频片段对象。请读者阅读下面的音频片段接口 AudioClip 说明文档。

java. applet. **AudioClip** 接口说明文档			
public interface **AudioClip**			
	修饰符	接口成员(节选)	功能说明
1		void **play**()	播放音频
2		void **stop**()	停止播放
3		void **loop**()	循环播放
		…	

Java 后来推出的 swing 框架也对 awt 的小应用程序类 Applet 进行了扩展,定义了一个新的小应用程序类 **JApplet**。JApplet 是 Applet 的子类,两者的用法基本相同。JApplet 与 Applet 的最大区别是,使用 Applet 编写小应用程序时只能使用旧的 awt 图形组件,而新的 JApplet 则可以使用新的 swing 图形组件。JApplet 与 JFrame、JDialog 一样,它们是 swing 框架中的 3 大顶层容器。

Java 小应用程序是一种早期用于增强 HTML 网页表现力的技术。现在,HTML 本身也在不断发展,其表现能力不断提高,例如 HTML 5。另外,JavaScript 脚本语言也可以在很大程度上替代 Java 小应用程序。目前,Java 小应用程序已经很少使用了,读者只需简单了解即可。**注**:全国计算机等级考试二级 Java 语言程序设计考试大纲(2018 年版)中还包含 Java 小应用程序(Applet)这个知识点。

本节习题

1. Java 小应用程序类 Applet 定义在 Java API 包(　　)当中。
   A. java. awt　　B. java. applet　　C. javax. swing　　D. java. util
2. Java 小应用程序类 JApplet 定义在 Java API 包(　　)当中。
   A. java. awt　　B. java. applet　　C. javax. swing　　D. java. util
3. 执行 Java 小应用程序时首先会调用其中的(　　)方法。
   A. init()　　B. start()　　C. stop()　　D. destroy()
4. Java 小应用程序类 Applet 中加载图像文件的方法是(　　)。
   A. init()　　B. start()
   C. getImage()　　D. getAudioClip()
5. Java 小应用程序类 Applet 中加载音频文件的方法是(　　)。
   A. init()　　B. start()
   C. getImage()　　D. getAudioClip()

本章学习要点

- 了解 Java API 中各图形组件之间的关系。
  - ◇ 框架窗口 JFrame 和对话框 JDialog 是顶层容器,其中包含内容面板。
  - ◇ 可以在内容面板中添加组件,并可设置不同的布局管理策略。

◇ 内容面板可使用JPanel划分出子面板,子面板独立布局,可实现比较复杂的图形界面。

- 了解Java图形用户界面程序的事件响应机制。
- 通过编程练习掌握常用组件的用法,并能根据程序功能要求设计图形用户界面。
- 在掌握上述图形用户界面基本编程原理之后,可通过Java API文档自行研究javax.swing包中其他各种不同功能的图形组件,例如JSplitPane、JTabbedPane、JEditorPane、JPasswordField、JPopupMenu、JToolBar、JToolTip、JProgressBar、JScrollBar、JSlider、JSpinner、JTree等。

本章习题

1. 编写程序。模仿6.2.1节例6-2的Java演示程序,先在窗口中绘制一个椭圆,然后在椭圆中显示问候语"Hello, World!"。编写实现上述功能的Java程序。

2. 编写程序。编写一个如本章开篇中图6-2所示的图形用户界面温度换算程序。

3. 重写程序。阅读并理解6.4.4节例6-10中单选按钮类JRadioButton的Java演示程序,然后重写这个程序。

4. 重写程序。阅读并理解6.5.1节例6-16中模式对话框的Java示例程序,然后重写这个程序。

5. 重写程序。阅读并理解6.6.1节例6-21中响应底层鼠标和键盘事件的Java演示程序,然后重写这个程序。

第7章 输入输出流

Java 语言将程序中数据的输入输出过程看作是一种数据流动的过程。将提供输入数据的数据源称作**输入流**（input stream），将输出数据时的目的地称作**输出流**（output stream）。例如，键盘就是一个输入流，而显示器则是一个输出流。输入流、输出流可统称为**输入输出流**（I/O stream）。Java API 为数据的输入输出提供了一组输入输出流类。

本章将学习数据输入输出的基本原理，学会运用 Java API 提供的输入输出流类实现**标准输入输**出和**文件输入输出**功能，最后通过具体的程序实例来了解文本文件、图像文件和声音文件的基本处理方法。

7.1 Java 输入输出流

本节讲解 Java 程序中数据输入输出的基本原理，并介绍 Java API 为程序员提供的一组输入输出流类。

7.1.1 Java 程序的输入输出

程序的功能是数据处理。程序需要从输入设备接收原始数据，处理后再将处理结果输出到输出设备，如图 7-1 所示。一个 Java 程序可以从键盘接收用户输入的数据，也可以从硬盘文件中读取原始数据，甚至可以从网络上读取其他计算机传输过来的数据。Java 程序可以将处理后的结果输出到显示器，也可以输出到硬盘上的某个文件中，或者发送给网络上的其他计算机。

1. 输入流与输出流

Java 语言将数据的输入输出过程看作是一种数据流动的过程。站在 Java 程序的角度，键盘是一种输入数据时的数据源，它是一个**输入流**；显示器则是一种输出数据时的目的地，它是一个**输出流**。而硬盘文件和计算机网络则既可以当作输入流，从中读取数据；也可以当作输出流，向其中写入数据。

2. 标准 I/O 与文件 I/O

通常，将键盘输入、显示器输出称作标准输入输出，简称**标准 I/O**；将硬盘文件的输入、输出称作文件输入输出，简称**文件 I/O**。计算机网络的输入、输出分别对应网络上数据的接

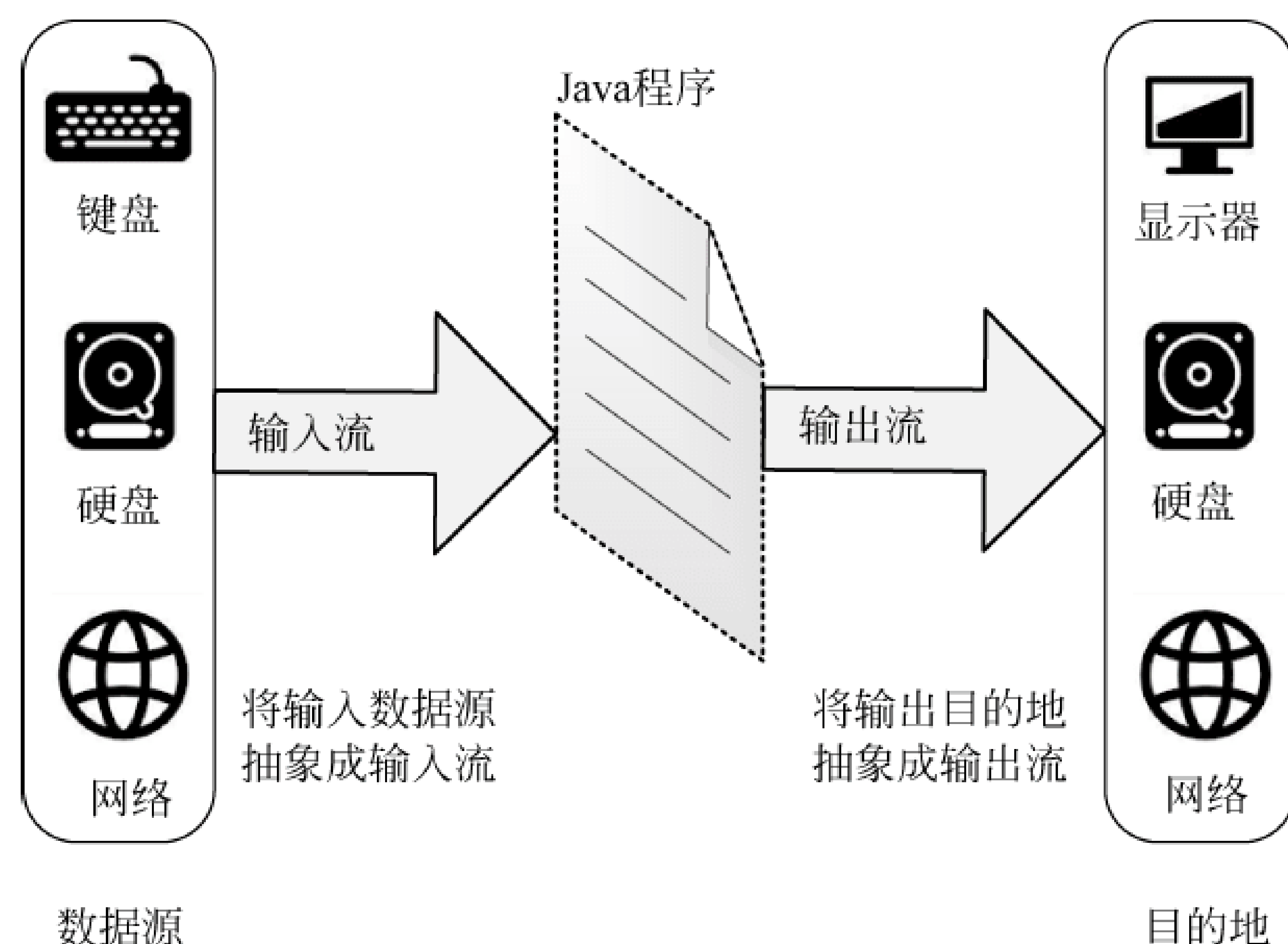


图 7-1 Java 程序的输入与输出

收和发送,它们被统称为**网络通信**。本章主要学习标准 I/O 和文件 I/O,网络通信将在第 9 章进行讲解。

3. 字节流与字符流

敲击键盘上的某个按键,键盘将输入一个该按键所对应字符的编码。例如敲击键盘上的字母 a,键盘将输入字符 a 所对应的 ASCII 编码,该编码值为 97(十进制)。若以十六进制表示则为 61,而其在计算机内部的存储格式为二进制的 01100001,占 1 字节。

在中文 Windows 操作系统上,假设通过键盘输入字符串"**abc 中国**",键盘将输入一个字符编码序列,以十六进制表示则该编码序列的内容如下:

```
61 62 63 d6 d0 b9 fa
```

在这个字符编码序列中,英文字符 abc 所对应的是其单字节 ASCII 编码,而中文字符"中国"对应的则是其双字节 GBK 编码。这种单字节、双字节混合编码方式被统称为 ANSI **编码**(**注**:中文 Windows 默认的系统编码是 ANSI 编码)。可以看出,在计算机内部,键盘输入的数据实际上是一个以**字节**(byte)为单位的数值序列。

站在 Java 程序的角度,可以将键盘输入的数据看作是一个以字节为单位的数据流。这种数据流称作字节型数据流,简称**字节流**(byte stream)。字节是计算机存储数据的基本单位,任何数据都可以看成由一个或多个字节组成的字节流。字节流是计算机内部的底层数据表示。它不对数据做任何解释,只是将数据单纯看作存放在字节里的一个个数。

Java 程序也可以将字节流数据看作是一个由字符编码组成的数据流。这种数据流称作字符型数据流,简称**字符流**(character stream)。字符流是在字节流的基础上对数据进行解释,将存放在字节里的数看作字符的编码,即字符流认为其中的数据都是文字字符。

4. 字节型输入输出流

字节型输入输出流将输入或输出的数据都当作 **byte** 型数值进行处理。换句话说,字节型输入输出流只能输入或输出 byte 型数值。其他类型数据需转换成 byte 型数值或数组才

能使用字节型输入输出流进行输入输出。字节型输入输出流包括**字节型输入流**和**字节型输出流**两种。

5. 字符型输入输出流

字符型输入输出流将输入或输出的数据都当作 **char** 型字符进行处理。换句话说，字符型输入输出流只能输入或输出 char 型字符。其他类型数据需转换成 char 型字符或数组（或 String 类型）才能使用字符型输入输出流进行输入输出。字符型输入输出流也包括**字符型输入流**和**字符型输出流**两种。

7.1.2 Java API 提供的输入输出流类

Java API 为程序员提供了一组输入输出流类，它们被定义在 java. io 包中。这组输入输出流类可划分为字节型和字符型两大类，每大类又可细分为输入流和输出流两个小类，总共 4 个类族。

- **字节型**输入输出流。
  - ◇ 字节型**输入流**类族。提供 byte 型数据的输入功能，根类是 **InputStream**。
  - ◇ 字节型**输出流**类族。提供 byte 型数据的输出功能，根类是 **OutputStream**。
- **字符型**输入输出流。
  - ◇ 字符型**输入流**类族。提供 char 或 String 类型数据的输入功能，根类是 **Reader**。
  - ◇ 字符型**输出流**类族。提供 char 或 String 类型数据的输出功能，根类是 **Writer**。

1. 输入输出流类说明文档

这里给出 4 个输入输出流类族根类的说明文档。请读者仔细阅读，了解其中输入或输出数据的方法。输入数据也称作**读**(read)数据，输出数据也作**写**(zwrite)数据。**注：**阅读文档时请对比字节型与字符型、输入流与输出流的不同之处，这样可以帮助理解、记忆。

java. io. **InputStream** 类说明文档（**字节型**输入流类族的根类）

public abstract class **InputStream**
extends **Object**
implements **Closeable**

	修　饰　符	类成员（节选）	功 能 说 明
1		**InputStream**()	构造方法
		int **available**()	返回字节输入流中可读的字节数（估计值）
2	abstract	int **read**()	从字节输入流中读出一个字节
3		int **read**(**byte**[] b)	根据数组 b 的长度读出若干个字节并存放到 b 中
4		int **read**(**byte**[] b, int off, int len)	读出 len 个字节并存放到数组 b 中
5		long **skip**(long n)	跳过 n 个字节
6		void **close**()	关闭字节输入流

…

java.io.**OutputStream** 类说明文档(**字节型**输出流类族的根类)			
public abstract class **OutputStream** extends **Object** implements **Closeable**,**Flushable**			
	修　饰	类成员(全部)	功能说明
1		**OutputStream**()	构造方法
2	abstract	void **write**(int b)	向字节输出流中写入一个字节
3		void **write**(**byte**[] b)	向字节输出流中写入数组 b
4		void **write**(**byte**[] b, int off, int len)	写入数组 b 中的 len 个字节
5		void **flush**()	立即输出缓存里的内容
6		void **close**()	关闭字节输出流
…			

java.io.**Reader** 类说明文档(**字符型**输入流类族的根类)			
public abstract class **Reader** extends **Object** implements **Readable**, **Closeable**			
	修　饰　符	类成员(节选)	功能说明
1	protected	**Reader**()	构造方法
2		boolean **ready**()	检查字符输入流是否有可读的字符
3		int **read**()	从字符输入流读出一个字符
4		int **read**(**char**[] cbuf)	根据字符数组 cbuf 的长度读出若干字符并存放到 cbuf 中
5	abstract	int **read**(**char**[] cbuf, int off, int len)	读出 len 个字符并存放到数组 cbuf 中
6		long **skip**(long n)	跳过 n 个字符
7	abstract	void **close**()	关闭字符输入流
…			

java.io.**Writer** 类说明文档(**字符型**输出流类族的根类)			
public abstract class **Writer** extends **Object** implements **Appendable**, **Closeable**,**Flushable**			
	修　饰　符	类成员(节选)	功能说明
1	protected	**Writer**()	构造方法
2		void **write**(int c)	向字符输出流写入一个字符
3		void **write**(**char**[] cbuf)	向字符输出流写入字符数组 cbuf
4	abstract	void **write**(**char**[] cbuf, int off, int len)	写入字符数组 cbuf 中的 len 个字符
5		void **write**(**String** str)	写入字符串 str
6		void **write**(**String** str, int off, int len)	写入字符串 str 中的 len 个字符
7	abstract	void **flush**()	立即输出缓存里的内容
8	abstract	void **close**()	关闭字符输出流
…			

2. 字节型输入流举例

Java API 在系统类 System 中预定义了一个表示键盘的静态字段 in,它就是字节型输入流类 InputStream 的对象。

```
static InputStream  in;     //系统类 System 中定义的表示键盘的静态字段 in
```

注: 类 InputStream 是一个抽象类,System. in 实际上是其子类的对象。

例 7-1 给出一个使用 System. in 对象进行键盘输入的 Java 演示程序,其中演示了字节型输入流类 InputStream 的使用方法。

例 7-1 使用 System. in 对象进行键盘输入的 Java 演示程序(JSystemInTest. java)

```
1  import java.io.*;                                //导入 java.io 包中的类
2  public class JSystemInTest {                     //测试类
3     public static void main(String[] args) {      //主方法
4         byte bbuf[] = new byte[20];               //定义字节数组保存键盘输入的字节流
5         //按字节流方式读取键盘输入的数据
6         try {                       //必须处理输入过程中可能抛出的勾选异常 IOException
7            int len = System.in.read(bbuf);  //此处可能会抛出 IOException 异常
8            for (int n = 0; n < len; n++) {
9               int x = Byte.toUnsignedInt( bbuf[n] );     //将字节流先转成无符号整数
10              String hexString = Integer.toHexString(x);  //再转成十六进制字符串
11              System.out.print(hexString + " ");          //依次显示输入的字节流
12           }
13           System.out.println();
14        }
15        catch(IOException e)                               //处理勾选异常 IOException
16        {  System.out.println( e.getMessage() );  }
17  } }
```

使用输入输出流进行输入输出时可能会抛出异常 **IOException**。这是一个勾选异常。Java 程序必须对勾选异常进行处理,否则编译不能通过。

在 Eclipse 集成开发环境中运行例 7-1 的程序,运行结果如图 7-2 所示。在图 7-2 中,从键盘输入字符串"abc 中国",程序将以十六进制显示从 System. in 这个字节型输入流对象中输入的数据。这些数据是字符串"abc 中国"在计算机内部的底层原始存储形式,其中最后两个十六进制数 d、a 分别是按 Enter 键时所输入的两个控制字符"回车"和"换行"的 ASCII 码。

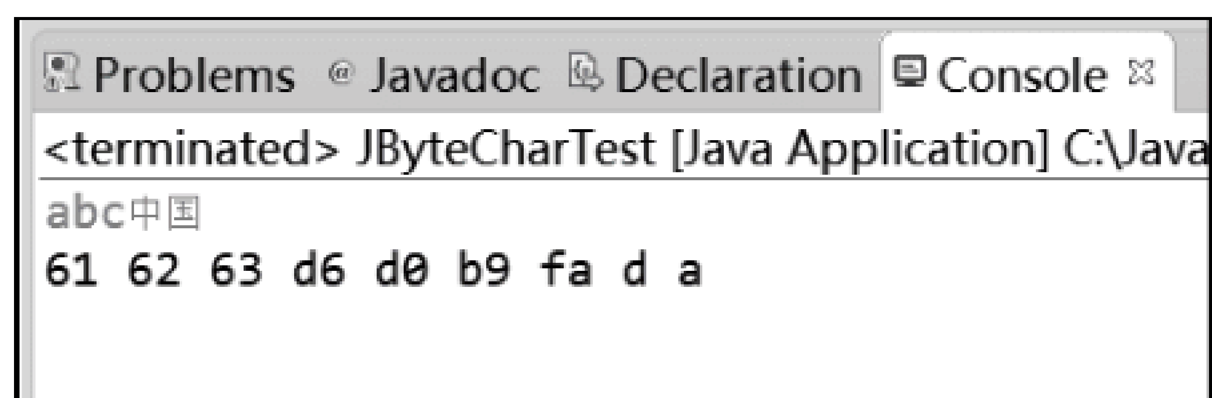


图 7-2 例 7-1 程序的运行结果(字节型输入流)

7.1.3 将字节流包装成字符流

Java API 还提供了两个将字节型输入输出流**包装**成字符型输入输出流的类，它们分别是输入流包装类 **InputStreamReader** 和输出流包装类 **OutputStreamWriter**。所谓包装类，就是将一个已有类的对象作为字段，然后调整或增强其功能。

例如，使用输入流包装类 InputStreamReader 可以将字节型输入流对象 System.in 包装成一个字符型输入流对象，其包装方法如下：

```
InputStreamReader inChar = new InputStreamReader( System.in);
```

其中，对象 **inChar** 就是包装后得到的字符型输入流对象。使用该对象可实现从键盘输入 char 型数据的功能。

输入流包装类 InputStreamReader 将字节型输入流里的底层原始数据看作字符的某种编码(默认为系统编码，通常为 ANSI 编码)，然后将该编码转换成 **Unicode 编码**(UTF-16)，这样就可以输入到 Java 语言的 char 型变量或数组或 String 类型中去了。例 7-2 给出一个使用输入流包装类 InputStreamReader 从键盘输入 char 型数据的 Java 演示程序。

例 7-2 一个使用输入流包装类 InputStreamReader 的 Java 演示程序(InputStreamReaderTest.java)

```
1   import java.io. * ;                                    //导入 java.io 包中的类
2   public class InputStreamReaderTest {                   //测试类
3       public static void main(String[ ] args) {          //主方法
4           char cbuf[ ] = new char[20];                   //定义字符数组保存键盘输入的字符流
5           try {                                          //必须处理勾选异常 IOException
6               InputStreamReader inChar = new InputStreamReader( System.in);  //包装
7               int len = inChar.read( cbuf );             //此处可能会抛出 IOException 异常
8               for (int n = 0; n < len; n++) {            //显示所输入的字符数据
9                   System.out.print(cbuf[n] + " ");       //字符之间用空格隔开
10              }
11              System.out.println();
12              inChar.close();                            //关闭输出流对象 inChar
13          }
14          catch(IOException e)                           //处理勾选异常 IOException
15          {   System.out.println( e.getMessage() );   }
16  }   }
```

在 Eclipse 集成开发环境中运行例 7-2 的程序，运行结果如图 7-3 所示。在图 7-3 中，从键盘输入字符串"abc 中国"，程序通过对字节型输入流对象 System.in 进行包装，将所接收到的底层字节流转换成 Java 语言的 char 型字符序列，然后再输入到字符数组中。依次显示数组中的各个字符，显示时以空格分隔。

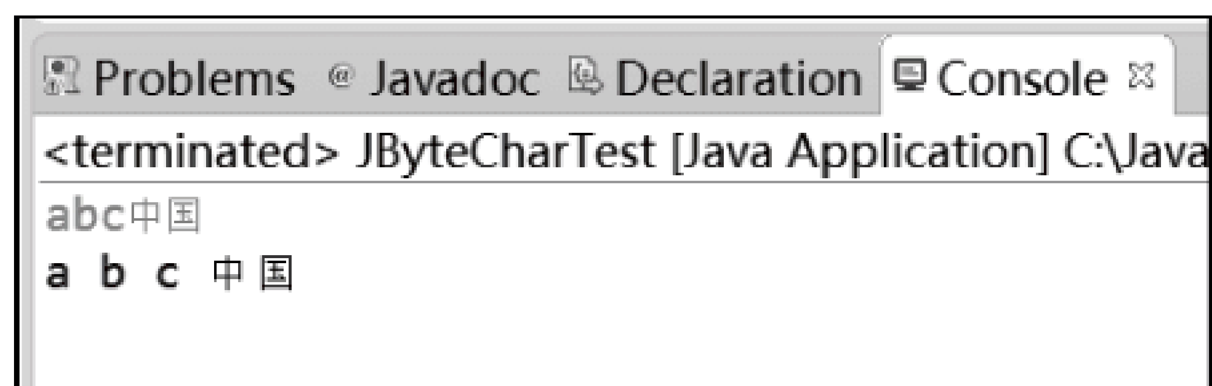


图 7-3 例 7-2 程序的运行结果(字符型输入流)

同样,输出流包装类 OutputStreamWriter 可以将字节型输出流对象包装成字符型输出流对象。使用包装后的字符型输出流对象,可以输出 Java 程序中的 char 或 String 类型数据。输出时,首先将 Java 语言的 Unicode 编码(UTF-16)转换成指定的编码格式(默认为系统编码,通常为 ANSI 编码),然后再将转换后的编码序列当作字节流,交由包装前的字节型输出流对象按字节执行底层的物理输出。

请读者阅读下面的输入流包装类 InputStreamReader 和输出流包装类 OutputStreamWriter 说明文档。

java. io. **InputStreamReader** 类说明文档(输入流包装类)

public class **InputStreamReader**
extends **Reader**

	修 饰 符	类成员(节选)	功 能 说 明
1		**InputStreamReader**(InputStream in)	构造方法(默认为系统编码)
2		**InputStreamReader**(InputStream in, String charsetName)	构造方法(指定字符编码)
3		String **getEncoding**()	获取字符编码类型
4		boolean **ready**()	检查字符输入流是否准备好
5		int **read**()	从字符输入流读出一个字符
6		int **read**(char[] cbuf, int off, int len)	读出 len 个字符并存放到数组 cbuf 中
7		void **close**()	关闭字符输入流

…

java. io. **OutputStreamWriter** 类说明文档(输出流包装类)

public class **OutputStreamWriter**
extends **Writer**

	修 饰 符	类成员(节选)	功 能 说 明
1		**OutputStreamWriter**(OutputStream out)	构造方法(默认为系统编码)
2		**OutputStreamWriter**(OutputStream out, String charsetName)	构造方法(指定字符编码)
3		String **getEncoding**()	获取字符编码类型
4		void **write**(int c)	向字符输出流写入一个字符
5		void **write**(char[] cbuf, int off, int len)	写入字符数组 cbuf 中的 len 个字符
6		void **write**(String str, int off, int len)	写入字符串 str 中的 len 个字符
7		void **flush**()	立即输出缓存里的内容
8		void **close**()	关闭字符输出流

…

图 7-4 给出 4 个输入输出流根类和 2 个包装类之间的继承和包装关系图。其中的 4 个输入输出流类都是根类,而且还是抽象类,不能直接创建输入输出流对象。Java API 在这 4 个根类的基础上不断进行扩展或包装,为程序员提供了丰富的输入输出功能,可满足各种不同的应用需求,例如标准 I/O、文件 I/O 等。

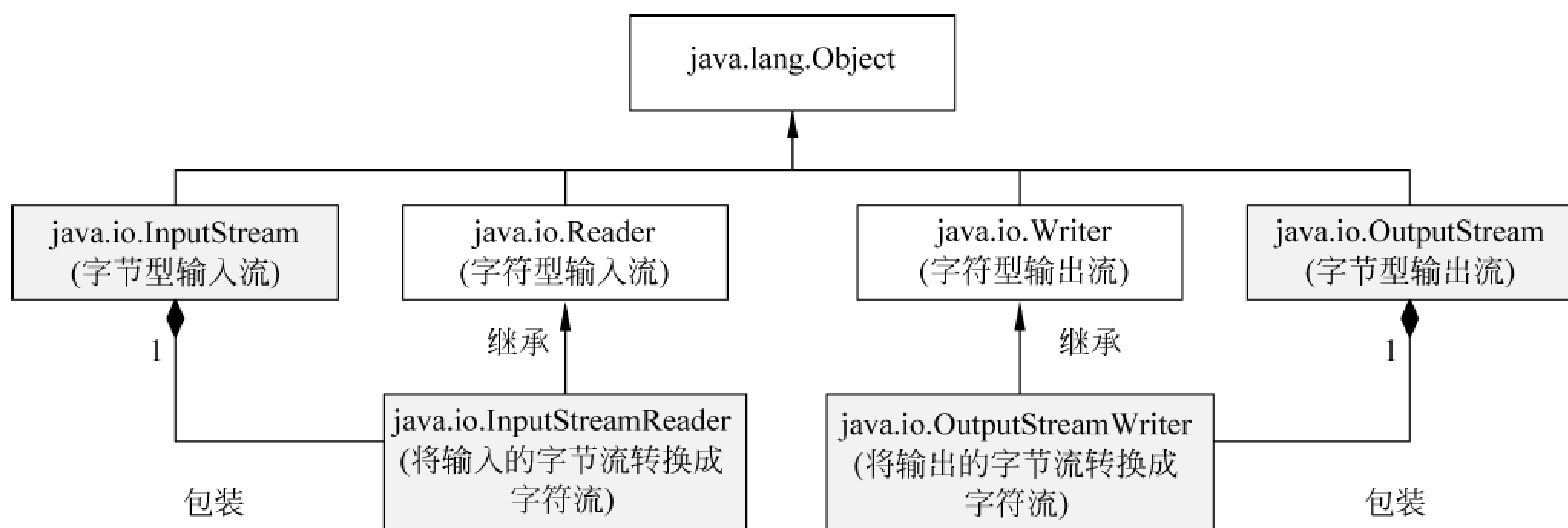


图 7-4　4 个输入输出流根类和 2 个包装类之间的继承和包装关系图

本节习题

1. Java 程序不能从(　　)输入数据。
   A. 键盘　B. 硬盘文件　C. 计算机网络　D. 显示器
2. Java 程序不能向(　　)输出数据。
   A. 键盘　B. 硬盘文件　C. 计算机网络　D. 显示器
3. 在 Java 语言中,(　　)不属于输入输出流的范畴。
   A. 标准 I/O　B. 文件 I/O
   C. 网络通信　D. 编辑源程序
4. 字节型输入流类 InputStream 可以将输入数据保存到(　　)数组中。
   A. byte[]　B. int[]　C. double[]　D. char[]
5. 字符型输出流类 Writer 可以输出保存在(　　)数组中的数据。
   A. byte[]　B. int[]　C. double[]　D. char[]
6. 输入流类中输入数据的方法是(　　)。
   A. read()　B. skip()　C. write()　D. close()
7. 输出流类中输出数据的方法是(　　)。
   A. read()　B. skip()　C. write()　D. close()
8. 输入输出流类中关闭数据流的方法是(　　)。
   A. read()　B. skip()　C. write()　D. close()
9. 将字节型输入流包装成字符型输入流的类是(　　)。
   A. InputStream　B. Reader
   C. InputStreamReader　D. OutputStreamWriter
10. 将字节型输出流包装成字符型输出流的类是(　　)。
   A. OutputStream　B. Writer
   C. InputStreamReader　D. OutputStreamWriter

7.2 标准 I/O

标准 I/O 指的是键盘输入,显示器输出,有时也被作**控制台**(console)输入输出。

键盘可以一次输入多个数据,多个数据之间以空格或 Tab 键等隔开。Java 程序需要对从键盘输入的数据进行扫描、分割,然后转换成指定的数据类型,例如转换成 int 型或 double 型,最后再将转换后的数据赋值给变量,这种输入方式称作**格式化输入**。格式化输入就是从字符流中输入数据。Java API 为格式化输入专门提供了一个扫描器类 **Scanner**。

Java 程序在显示器上输出数据时,需要将程序中不同类型的数据统一转换成字符串,转换时还需要指定显示格式,例如显示宽度或保留几位小数等,然后再输出到显示器上,这种输出方式称作**格式化输出**。格式化输出就是将不同类型的数据格式化成字符流,然后再进行输出。Java API 为格式化输出专门提供了一个打印流类 **PrintStream**。

7.2.1 格式化输入

Java API 提供的扫描器类 **Scanner** 可以实现键盘的格式化输入功能。扫描器类 Scanner 是一个包装类,可以对字节型输入流对象 **System.in** 进行包装,包装得到的键盘扫描器对象具有格式化输入功能。扫描器类 Scanner 还可以从硬盘文件或从一个字符串中进行格式化输入。

例 7-3 给出一个使用扫描器类 Scanner 进行格式化输入的 Java 演示程序。

例 7-3 使用扫描器类 Scanner 进行格式化输入的 Java 演示程序(JScannerTest.java)

```
1  import java.io.*;                          //导入 java.io 包中的类
2  import java.util.Scanner;                  //导入 java.util 包中定义的扫描器类 Scanner
3
4  public class JScannerTest {                //测试类
5      public static void main(String[] args) { //主方法
6          Scanner sc = null;                 //定义一个扫描器引用变量
7          System.out.print( "从键盘输入数据: " );
8          try {          //输入时可能会抛出异常,使用扫描器类建议按此代码框架编写程序
9              sc = new Scanner( System.in );  //将键盘对象 System.in 包装成扫描器对象
10             String str = sc.next();        //读取下一个字符串
11             int x = sc.nextInt();          //读取下一个 int 型整数
12             System.out.println( "输入结果为: " + str + ", " + x );//显示输入结果
13         }
14         finally
15         {  if (sc != null)  sc.close();  }//关闭扫描器
16 }  }
```

在 Eclipse 集成开发环境中运行例 7-3 的程序,在键盘上输入一个字符串和一个整数,程序将显示扫描器对象的输入结果。例如:

从键盘输入数据:**abcd 123** <回车键>

输入结果为：abcd，123

请读者阅读下面的扫描器类 Scanner 说明文档。

java. util. **Scanner** 类说明文档			
public final class **Scanner** extends **Object** implements **Iterator**<**String**>，**Closeable**			
	修　饰　符	类成员(节选)	功 能 说 明
1		**Scanner**(InputStream source)	构造方法(从字节输入流输入)
2		**Scanner**(InputStream source,String charsetName)	构造方法(指定字符编码)
3		**Scanner**(Path source)	构造方法(从文件输入)
4		**Scanner**(Path source, String charsetName)	构造方法(指定字符编码)
5		**Scanner**(File source)	构造方法(从文件输入)
6		**Scanner**(String source)	构造方法(从字符串输入)
7		boolean **hasNext**()	检查扫描器中是否还有输入项
8		String **next**()	读出下一个字符串
9		int **nextInt**()	读出下一个 int 型整数
10		double **nextDouble**()	读出下一个 double 型浮点数
11		Scanner **useDelimiter**(String pattern)	设置数据项的分隔符
12		void **close**()	关闭扫描器
		…	

7.2.2 格式化输出

Java API 提供了一个可以实现格式化输出功能的字节型打印流类 **PrintStream**，其中两个最主要的方法成员是 **println**()和 **format**()。

打印流类 PrintStream 是字节型输出流类 OutputStream 的二级子类。使用打印流类 PrintStream 可以将数据格式化，然后再输出到显示器上或硬盘文件中。请读者阅读下面的字节型打印流类 PrintStream 说明文档。

java. io. **PrintStream** 类说明文档			
public class **PrintStream** extends **FilterOutputStream** implements **Appendable**，**Closeable**			
	修　饰　符	类成员(节选)	功 能 说 明
1		**PrintStream**(OutputStream out)	构造方法(向字节输出流输出)
2		**PrintStream**(String fileName)	构造方法(向文件输出)
3		**PrintStream**(String fileName, String csn)	构造方法(指定字符编码)
4		**PrintStream**(File file)	构造方法(向文件输出)
5		void **println**(String x)	格式化输出一个字符串
6		void **print**(String s)	格式化输出(不换行)

续表

	修　饰　符	类成员(节选)	功能说明
7		void **println**(int x)	格式化输出一个 int 型整数
8		void **print**(int x)	格式化输出(不换行)
9		void **println**(double x)	格式化输出一个 double 型实数
10		void **print**(double x)	格式化输出(不换行)
11		void **println**(Object x)	格式化输出一个对象
12		void **print**(Object obj)	格式化输出(不换行)
13		void **println**()	单独换一行
14		PrintStream **format**(String format,Object... args)	按照格式化字符串输出多个对象
15		PrintStream **printf**(String format, Object... args)	同 format()函数,类似于 C 语言的 printf()函数
16		void **write**(byte[] buf, int off, int len)	输出字节数组
17		void **flush**()	立即输出缓存里的内容
18		void **close**()	关闭打印流
		...	

Java API 在系统类 System 中预定义了一个表示显示器的静态字段 out,它就是打印流类 PrintStream 的对象。

```
static PrintStream  out;  //系统类 System 中定义的表示显示器的静态字段 out
```

程序员直接使用这个 System.out 对象,就可以在显示器上格式化输出数据。

例 7-4 给出一个使用 System.out 对象在显示器上进行格式化输出的 Java 演示程序,其中演示了 println()和 format()的不同用法。

例 7-4　一个使用 System.out 对象在显示器上进行格式化输出的 Java 演示程序(JPrintStreamTest.java)

```
 1  import java.io.*;                                    //导入 java.io 包中的类
 2  public class JPrintStreamTest {                      //测试类
 3      public static void main(String[] args) {         //主方法
 4          int x = 10;                                  //先定义几个演示数据
 5          double y = 15.8;
 6          String str = "abcd";
 7          //下面演示格式化输出方法
 8          System.out.println( x);                      //输出一个 int 型整数
 9          System.out.println( y);                      //输出一个 double 型实数
10          System.out.println( str);                    //输出一个 String 字符串
11          //下面演示同时格式化输出多个数据的方法
12          System.out.println("x = " + x + ", y = " + y + "str = " + str);
                                                         //输出一个字符串表达式
13          System.out.format("x = %d, y = %f, str = %s\n", x, y, str);
                                                         //格式化输出多个数据项
14          System.out.format("x = %h\n", x);            //显示十六进制整数
15          System.out.printf("x = %d, y = %f, str = %s\n", x, y, str);
                                                         //类似于 C 语言的 printf()函数
16  } }
```

在 Eclipse 集成开发环境中运行例 7-4 的程序,运行结果如图 7-5 所示。请读者对照图 7-5 的显示结果来理解 println()和 format()的用法。

```
Problems  Javadoc  Declaration  Console
<terminated> JPrintStreamTest [Java Application] C:\Ja
10
15.8
abcd
x=10, y=15.8, str=abcd
x=10, y=15.800000, str=abcd
x=a
x=10, y=15.800000, str=abcd
```

图 7-5 例 7-4 程序的运行结果

本节习题

1. 扫描器类 Scanner 中输入 int 型整数的方法是(　　)。
   A. next()　　B. nextInt()　　C. nextDouble()　　D. hasNext()
2. 扫描器类 Scanner 中输入字符串数据的方法是(　　)。
   A. next()　　B. nextInt()　　C. nextDouble()　　D. hasNext()
3. 扫描器类 Scanner 中检查是否还有下一个输入数据的方法是(　　)。
   A. next()　　B. nextInt()　　C. nextInt()　　D. hasNext()
4. Java API 中具有格式化输入功能的类是(　　)。
   A. InputStream　　B. Reader
   C. InputStreamReader　　D. Scanner
5. System.in 是(　　)类的对象。
   A. InputStream　　B. Reader
   C. InputStreamReader　　D. Scanner
6. Java API 中具有格式化输出功能的类是(　　)。
   A. OutputStream　　B. Writer
   C. OutputStreamWriter　　D. PrintStream
7. 打印流类 PrintStream 中在格式化输出时会自动换行的方法是(　　)。
   A. println()　　B. print()　　C. format()　　D. printf()
8. System.out 是(　　)类的对象。
   A. OutputStream　　B. Writer
   C. OutputStreamWriter　　D. PrintStream

7.3 文件及文件 I/O

除了从键盘输入数据,用户还能以文件形式向程序批量输入原始数据。例如,某个气象观测站的观测员在记录每天的气温数据时,可以使用 Windows“记事本”程序并按如下格式

将一个月的气温数据输入计算机。

```
1      30.2
2      28.7
…
30     25.9
```

可以将上述气温数据保存成一个硬盘文件，例如保存到文本文件 temperature.txt 中。

可以编写一个 Java 程序来分析气温数据，如求月最高气温、最低气温或平均气温等。这时，Java 程序需要直接从硬盘文件中读取原始数据，这被称为**文件输入**。程序也可能需要将处理结果写入硬盘文件，这被称为**文件输出**。输出到显示器上的处理结果只能实时观看。程序一旦退出，所有显示结果都将丢失。而将处理结果保存成外存（例如硬盘、U 盘等）上的文件，这些外存文件可以长期保存，也可以复制或传送给其他人。

文件的输入输出被简称为**文件 I/O**。本节将介绍文件的基本概念，并具体讲解如何利用 Java API 中的输入输出流类实现文件的输入输出操作。

7.3.1 文件的基本概念

按存储内容分，计算机中的文件可划分成两大类：一类是**程序文件**；另一类是**数据文件**。程序文件存储的是某个程序的可执行代码，数据文件存储的则是某个程序生成的结果数据。例如，在一台安装了 Windows 操作系统的计算机上会有一个名为 notepad.exe 的文件，该文件就是保存“记事本”程序代码的程序文件。执行“记事本”程序，输入文字内容，然后保存到一个硬盘文件中。这个由“记事本”程序所创建的文件就是一个数据文件（扩展名为.txt）。再如，Word 文字处理程序在安装后被存放在硬盘上的某个程序文件中，通常其文件名为 WINWORD.exe。由 Word 文字处理程序所创建的 Word 文档则属于数据文件（扩展名为.doc 或.docx）。

这里对文件 I/O 做一个**限定**：本章所说的文件 I/O 指的是数据文件的输入输出。文件 I/O 将介绍 Java 程序如何从数据文件中输入数据，以及如何向数据文件输出数据。

1. 文件名

计算机以**文件**（file）为单位来管理存储在外存（硬盘、光盘、U 盘等）上的信息。当文件数量很多时，可以为文件建立分类**目录**（directory），将文件分散在不同目录下进行管理。目录下可以再建立**子目录**（subdirectory）。同一子目录下的文件不能重名。文件所在的目录称为该文件的**路径**（path）。图 7-6 给出了一个 Windows 操作系统的目录结构示意图。Windows 操作系统将目录也称作**文件夹**（folder）。

文件名（file name）是文件的唯一标识。不同操作系统的文件名格式有一些区别。在 Windows 系统中，一个完整的文件名格式为：

盘符:\目录名\子目录名\……\文件名.扩展名

其中，盘符、目录、子目录和文件名之间都用反斜杠“\”隔开，文件名和扩展名之间用点“.”隔开。例如，图 7-6 中“记事本”程序文件的完整文件名为：

```
C:\Windows\notepad.exe
```

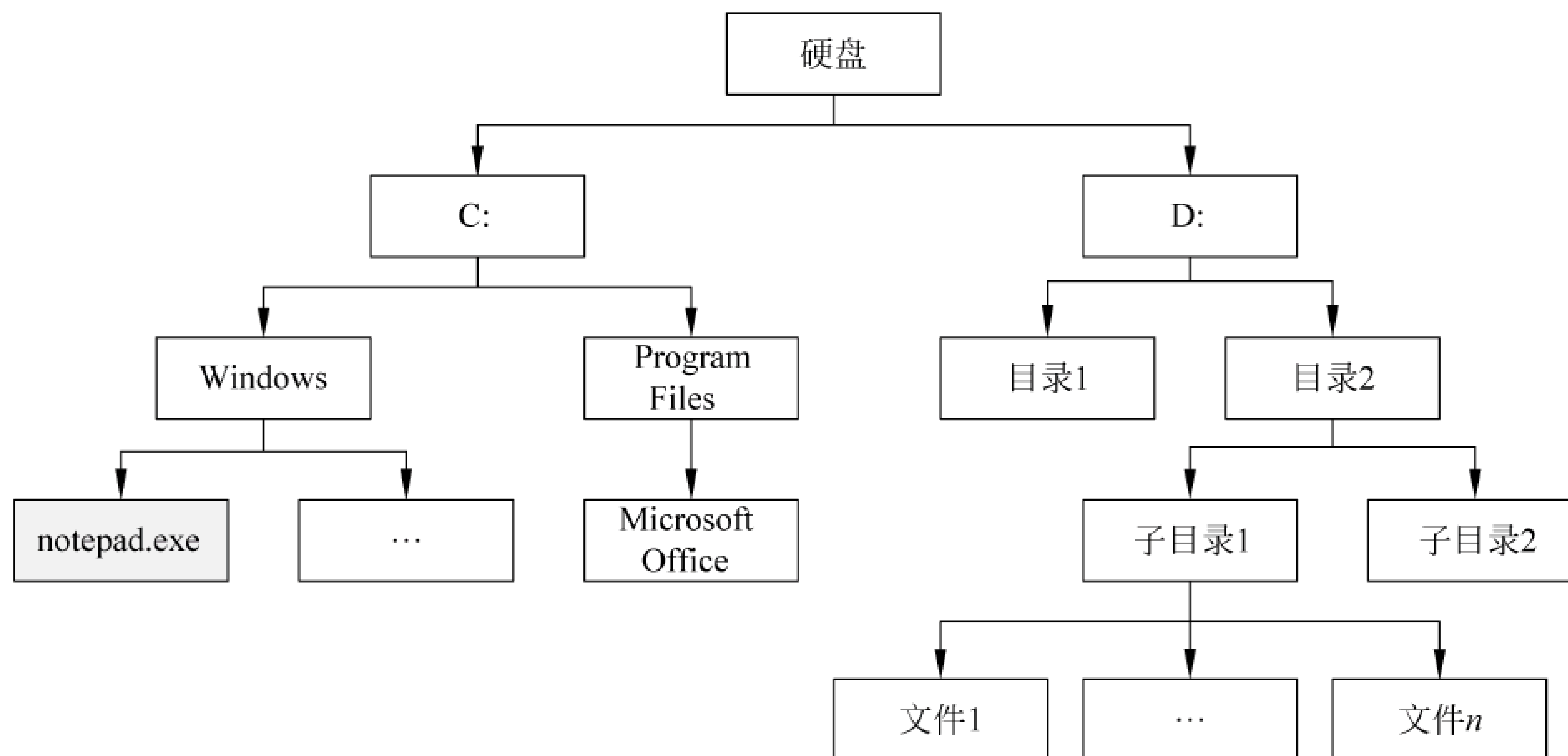


图 7-6　一个 Windows 操作系统的目录结构示意图

在 Java 源程序中，如果用字符串常量的形式来书写"记事本"程序的文件名，则应写成如下形式：

```
"C:/Windows/notepad.exe"      //正斜杠
```

或

```
"C:\\Windows\\notepad.exe"   //字符串常量中的反斜杠应使用转义字符的形式
```

2. 文件格式

按存储格式分，计算机中的数据文件可划分成两大类：一类是**文本文件**(text file)；另一类是**二进制文件**(binary file)。

1) 文本文件

文本文件用于存储字符类型的数据，并且主要是可见字符，例如英文字母、阿拉伯数字、标点符号，以及其他语种的文字字符(如中文字符)等。文本文件具有如下特点。

- **存储字符编码**。文本文件存储的内容是字符序列。存储字符就是存储字符的编码，例如，英文字母存储的是其 ASCII 编码(单字节)，中文字符存储的是 GBK 编码(双字节)。
- **具有换行格式**。文本换行时，存储两个控制字符 CR(ASCII 编码为 13)和 LF(ASCII 编码为 10)。**注**：不同操作系统可能会有所不同，例如 UNIX 和 Linux 操作系统的文本文件换行时只存储一个控制字符 LF。
- **通用性强**。文本文件存储的是纯文本内容，而且使用的是标准编码。文本文件不含任何其他附加信息(例如字体、排版格式等)。阅读文本文件不需要安装特殊的软件，使用类似于"记事本"这样的常规软件就能阅读、修改。换句话说，文本文件的通用性强。
- **可用于数据交换**。文本文件通用性强，可用于数据交换。例如，一个程序的处理结果可以通过文本文件输出给人来阅读，这是"**程序-人**"之间的数据交换；一个程序的

处理结果可以通过文本文件输入给另一个程序，这是“**程序-程序**”之间的数据交换。

2）二进制文件

如果只是程序与程序之间交换数据，那么除了文本文件之外还可以使用二进制文件。二进制文件是直接以内存的二进制存储格式在外存上存储数据。换句话说，二进制文件中数据的存储格式与该数据在内存中的存储格式是一致的。计算机内存以二进制存储数据，存储时还涉及占用字节数、整数格式或浮点格式等存储格式。不同数据类型具有不同的存储格式。使用二进制文件保存内存变量中的数据，就是不经过任何格式转换，直接将其内存单元的内容复制到外存文件中。和文本文件相比，二进制文件具有如下特点。

- **可保存任意类型的数据**。文本文件只存储字符类型数据，任何其他类型的数据必须转换成字符串才能保存到文本文件中。而二进制文件可以直接保存任意类型的数据。
- **存储效率高**。和文本文件相比，将内存变量中数据保存成二进制文件的效率更高。效率高具体表现在两个方面：一是不需要格式转换，保存速度快；二是保存数值型数据所占用的存储空间少。例如，保存一个 short 型整数－2100，使用二进制文件只需 2 字节。而使用文本文件则需将 short 型整数－2100 转换成字符串"－2100"，保存该字符串需要 5 字节。
- **通用性差**。二进制文件是程序员自己定义的一种私有格式。不同程序会创建不同的二进制文件。不管什么类型的数据，保存为二进制文件后都变成了二进制的 0、1 序列。二进制文件天生就是一种加密的文件。在不了解存储格式的情况下，任何程序都无法正确解释由其他程序创建的二进制文件。和文本文件相比，二进制文件的通用性差。对于二进制数据文件，通常是由哪个程序创建，就由哪个程序负责阅读、修改。
- **交换数据需遵循相同的格式标准**。为了能在不同程序间通过二进制文件交换数据，人们需要为二进制文件制定共同的格式标准。例如为了交换图像数据，人们专门制定了一些二进制图像文件格式标准，常用的有 JPEG、BMP、GIF 和 TIFF 等。

7.3.2　文件类 File

Java API 提供了一个文件类 **File**，用于描述外存上的文件或目录信息。使用文件类 File 可以获取文件信息、修改文件名或删除文件。使用文件类 File 也可以创建或删除目录。请读者阅读下面的文件类 File 说明文档。

java.io.**File** 类说明文档

public class **File**
extends **Object**
implements **Serializable**，**Comparable** < File >

	修　饰　符	类成员（节选）	功 能 说 明
1		**File**(String pathname)	构造方法
2		**File**(URI uri)	构造方法
3		**File**(File parent, String child)	构造方法
4		boolean **exists**()	检查文件或目录是否存在
5		String **getPath**()	获取路径

续表

	修 饰 符	类成员(节选)	功 能 说 明
6		String **getName**()	获取文件名
7		String **getAbsolutePath**()	获取完整的路径(绝对路径)
8		boolean **isFile**()	是否是普通文件
9		boolean **isDirectory**()	是否是目录
10		long **length**()	返回文件长度
11		boolean **mkdirs**()	创建目录
12		boolean **delete**()	删除文件或目录
13		boolean **renameTo**(File dest)	重命名
14		boolean **canExecute**()	是否可执行
15		boolean **canRead**()	是否可读
16		boolean **canWrite**()	是否可写
17		String[] **list**()	返回目录下的所有文件和子目录
18		long **getFreeSpace**()	获取分区的空闲空间
		…	

7.3.3 文本文件 I/O

文本文件 I/O 所存储的是字符类型数据。Java API 为字符类型数据的输入输出提供了两个流类族(见图 7-7)。

- 字符型**输入流**类族,提供 char 或 String 类型数据的输入功能,根类是 **Reader**。
- 字符型**输出流**类族,提供 char 或 String 类型数据的输出功能,根类是 **Writer**。

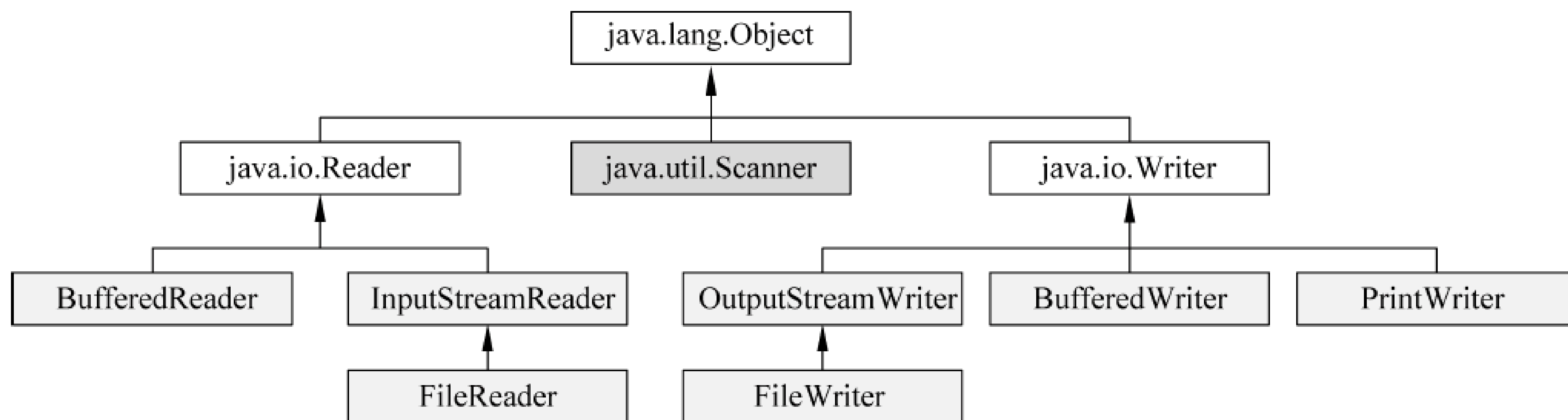


图 7-7 两个字符型输入输出流类族(未包含包名的类都属于 java.io 包)

为了演示文本文件的输入输出操作,本节先创建一个文本文件,向其中写入(输出)一段文字内容,然后再读出(输入)这个文本文件中的内容并显示到显示器上,检查所读出的内容是否正确。

可以使用字符型**文件**输出流类 **FileWriter** 来创建文本文件并向其中写入文字内容,使用字符型**文件**输入流类 **FileReader** 来读取文本文件中的内容。例 7-5 给出一个完整的文本文件输入输出演示程序。阅读例 7-5 的程序代码,读者可以了解文本文件输入输出的全过程,并初步领会 FileWriter 和 FileReader 这两个文件输入输出流类的使用方法。

例 7-5 一个完整的文本文件输入输出演示程序(JFileWRTest.java)

```
1  import java.io.*;                                        //导入 java.io 包中的类
2  public class JFileWRTest {           //测试类：演示文件流类 FileWriter 和 FileReader 的用法
3      public static void main(String[] args) {            //主方法
4          fwrite("d:/fwr.txt");        //调用下面的方法 fwrite()：创建并输出一个文本文件
5          fread("d:/fwr.txt");  //再调用下面的方法 fread()：输入并显示文本文件中的内容
6      }
7      static void fwrite(String fileName) {                //创建并输出文本文件的方法
8          try {                                            //必须处理勾选异常 IOException
9              FileWriter fw = new FileWriter( fileName);      //创建字符型文件输出流
10             fw.write("中国农业大学作为教育部直属高校,\r\n");      //向文件写入数据
11             fw.write("其历史源自于 1905 年成立的京师大学堂农科大学.\r\n");
12             fw.write("1949 年 9 月,由三所大学下属的农学院合并而成……");
13             fw.close();                                  //关闭文件输出流
14             System.out.println(fileName + "输出成功.");
15             System.out.println();
16         }
17         catch( IOException e)                            //处理 IOException 异常(勾选异常)
18         {   System.out.println( e.getMessage() );   }
19     }
20     static void fread(String fileName) {                 //输入文本文件并显示到显示器的方法
21         try {                                            //必须处理勾选异常 IOException
22             FileReader fr = new FileReader( fileName);      //创建字符型文件输入流
23             int c;
24             while ( (c = fr.read()) != -1 ) {   //从文件读取字符,读完为止
25                 System.out.print( (char)c );     //显示所读出的字符 c
26             }
27             fr.close();                                  //关闭文件输入流
28             System.out.println();
29             System.out.println(fileName + "输入成功.");
30         }
31         catch( IOExceptione)                             //处理 IOException 异常(勾选异常)
32         {   System.out.println( e.getMessage() );   }
33 } }
```

文本文件 I/O 的编程要点如下。

(1) **字符型文件输出流类** FileWriter。这个类的功能是创建文本文件,并向其中写入文字内容。创建 FileWriter 类的对象,创建时需指定文件名。创建 FileWriter 类对象,将自动在外存上创建一个文本文件。如果所创建的文件已存在,则先删除该文件,再创建一个新的空文件。使用 FileWriter 类的方法成员 **write**(),可以向文件文件中写入 char 或 String 类型的文字内容。

(2) **字符型文件输入流类** FileReader。这个类的功能是读取文本文件中的内容。创建 FileReader 类的对象,创建时需指定文件名。创建 FileReader 类对象,将自动打开外存上的

文本文件。如果所指定的文件不存在，则抛出一个 **FileNotFoundException** 异常(IOException 的子类)。

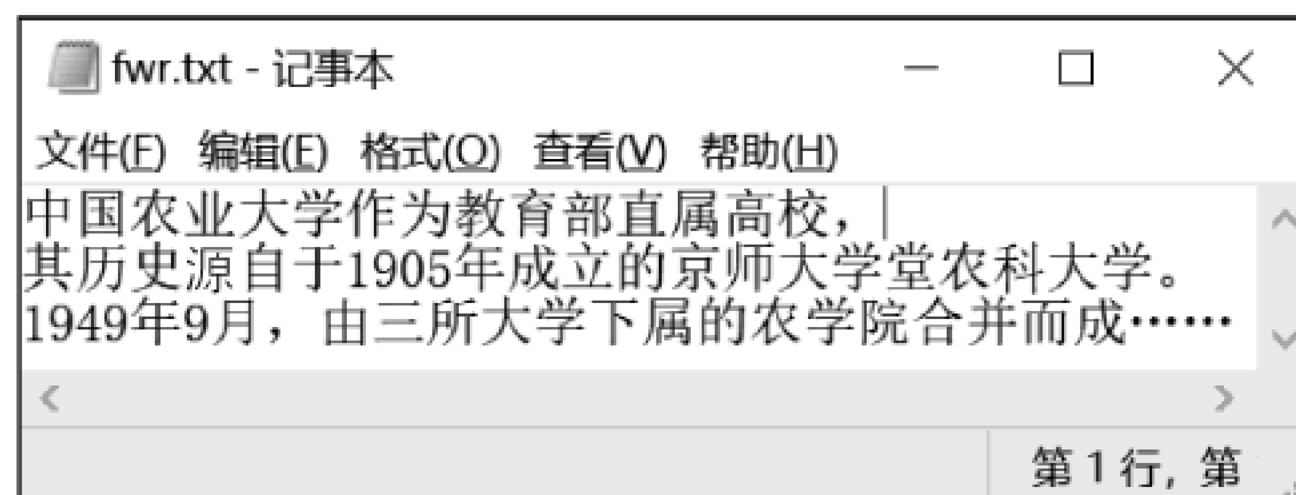
(3) **异常处理**。所有对文件的输入输出操作都可能会抛出 **IOException** 类或其子类的异常。它们属于勾选异常，Java 程序必须捕获并处理这些异常。程序员应当按照例 7-5 的代码框架来编写文件 I/O 程序。

(4) **关闭输入输出流**。文件 I/O 操作结束后，程序员应当调用文件输入输出流类的方法成员 **close**()，关闭文件输入输出流。通常情况下，文件只能被一个程序操作，这样可以避免读写冲突。

在 Eclipse 集成开发环境中运行例 7-5 的程序，运行结果如图 7-8(a)所示。可以使用其他程序，例如 Windows 的记事本程序，打开例 7-5 所创建的文本文件 fwr. txt，比对检查文本文件输入输出的内容是否正确，参见图 7-8(b)。



(a) 例7-5程序的运行结果



(b) 使用记事本程序打开例7-5程序所创建的文本文件fwr.txt

图 7-8　比对检查文本文件输入输出的内容是否正确

7.3.4　带缓冲区的文本文件 I/O

使用内存**缓冲区**(buffer)可以提高外存文件的读写速度。例如，假设需要将 100 个数据写入外存文件，可以每次只向文件写入一个数据，总共写 100 次；也可以先将 100 个数据写入某个内存缓冲区，再将该内存缓冲区的数据一次性写入文件。后者使用了内存缓冲区，其写入文件的速度比前者要快很多。

可以对字符型文件输入输出流类 FileReader、FileWriter 进行再次包装，将它们包装成**带缓冲区**的字符型输入流类 **BufferedReader** 和带缓冲区的字符型输出流类 **BufferedWriter**。具体的包装方法如下：

```
BufferedReader br = new BufferedReader( new FileReader(fileName) );   //包装 FileReader 对象
BufferedWriter bw = new BufferedWriter( new FileWriter(fileName) );   //包装 FileWriter 对象
```

使用包装后新的输入输出流对象，对文件进行读写操作时会自动使用内存缓冲区。例 7-6 给出了一个使用缓冲区进行文本文件 I/O 的 Java 演示程序。

例 7-6　一个使用缓冲区进行文本文件 I/O 的 Java 演示程序(JBufferedWRTest. java)

```
 1  import java.io.*;                          //导入 java.io 包中的类
 2  public class JBufferedWRTest {             //测试类
 3      public static void main(String[] args) { //主方法
 4          fwrite("d:/bwr.txt"); //调用下面的方法 fwrite(): 创建并输出一个文本文件
 5          fread("d:/bwr.txt");  //再调用下面的方法 fread(): 输入并显示文本文件中的内容
 6      }
 7      static void fwrite(String fileName) {  //创建并输出文本文件的方法
 8          try {                              //必须处理勾选异常 IOException
 9              //将字符型文件输出流对象包装成带缓冲区的字符型文件输出流对象
10              BufferedWriter bw = new BufferedWriter( new FileWriter(fileName) );
11              bw.write("中国农业大学作为教育部直属高校,");
12              bw.newLine();                  //BufferedWriter 增加了换行方法 newLine()
13              bw.write("其历史源自于 1905 年成立的京师大学堂农科大学.");
14              bw.newLine();
15              bw.write("1949 年 9 月,由三所大学下属的农学院合并而成……");
16              bw.close();                    //关闭文件输出流
17              System.out.println(fileName +"输出成功.");
18              System.out.println();
19          }
20          catch(IOException e)               //处理 IOException 异常(勾选异常)
21          {   System.out.println( e.getMessage() );   }
22      }
23      static void fread(String fileName) {    //输入文本文件并显示到显示器的方法
24          try {                              //必须处理勾选异常 IOException
25              //将字符型文件输入流对象包装成带缓冲区的字符型文件输入流对象
26              BufferedReader br = new BufferedReader( new FileReader(fileName) );
27              String s;
28              while ( (s = br.readLine()) != null ) {  //BufferedReader 增加了读一行的方
法
29                  System.out.print( s );     //所读出的字符串中去掉了回车和换行符
30                  System.out.println();
31              }
32              br.close();                    //关闭文件输入流
33              System.out.println(fileName +"输入成功.");
34          }
35          catch(IOException e)               //处理 IOException 异常(勾选异常)
36          {   System.out.println( e.getMessage() );   }
37  }   }
```

不同操作系统在文本文件的换行方法上存在一些差别。例如，Windows 操作系统使用两个控制字符 CR(ASCII 编码为 13)和 LF(ASCII 编码为 10)来表示换行，而 UNIX/Linux

操作系统只用一个控制字符 LF。

包装后,带缓冲区的字符型文件输出流类 BufferedWriter 增加了一个换行方法 **newLine()**,带缓冲区的字符型文件输入流类 BufferedReader 则增加了一个读一行的方法 **readLine()**。这两个方法能识别操作系统类型,自动对文本文件的换行做出不同处理。

例 7-6 程序的运行结果与例 7-5 完全相同,所不同的是例 7-6 在输入输出文件时会自动使用内存缓冲区,因此读写速度更快。

7.3.5 格式化文本文件 I/O

文本文件只能存储字符类型数据,即 char 或 String 类型数据。任何其他类型数据,例如 int 型或 double 型等,都需要经过**格式化**转换成字符流,即转换成字符串形式才能写入文本文件。

Java API 提供了一个**字符型**打印流类 **PrintWriter**,可以将各种不同类型的数据格式化成字符流,然后输出到文本文件。其使用方法与在显示器上进行格式化输出的方法基本相同(参见 7.2.2 节)。

注 1:字符型打印流类 PrintWriter 与字节型打印流类 PrintStream 的使用方法基本相同。

注 2:常用的显示器对象 System.out 是字节型打印流类 PrintStream 的对象。

使用扫描器类 Scanner 也可以对文本文件进行格式化输入。其使用方法与通过键盘进行格式化输入的方法基本相同(参见 7.2.1 节)。

例 7-7 给出一个对文本文件进行格式化输入输出的 Java 演示程序。

例 7-7 一个对文本文件进行格式化输入输出的 Java 演示程序(JFormatWSTest.java)

```
1  import java.io.*;                                    //导入 java.io 包中的类
2  import java.util.Scanner;                       //导入 java.util 包中定义的扫描器类 Scanner
3  public class JFormatWSTest {                         //测试类
4      public static void main(String[] args) {         //主方法
5          fprint("d:/pws.txt");       //调用下面的方法 fprint():创建并输出一个文本文件
6          fscan("d:/pws.txt");  //再调用下面的方法 fscan():输入并显示文本文件中的内容
7      }
8      static void fprint(String fileName) {            //创建并格式化输出文本文件的方法
9          try {                                        //必须处理勾选异常 IOException
10           PrintWriter pw = new PrintWriter( fileName );  //创建字符型文件打印流
11           int x = 10;  double y = 15.8;  String str = "abcd";
12           pw.print(x + " ");                         //输出一个 int 型整数
13           pw.print(y + " ");                         //输出一个 double 型实数
14           pw.println(str);                           //输出一个 String 字符串
15           pw.format("%d %f %s", x, y, str);  //格式化输出多个数据项
16           pw.println();
17           pw.close();                                //关闭文件打印流
18           System.out.println(fileName + "输出成功.");  System.out.println();
```

```
19            }
20            catch(IOException e)                //处理 IOException 异常(勾选异常)
21            {  System.out.println( e.getMessage() );   }
22        }
23        static void fscan(String fileName) {      //格式化输入文本文件并显示到显示器的方法
24            Scanner sc = null;                   //定义一个扫描器引用变量
25            try {                                //必须处理勾选异常 IOException
26                sc = new Scanner( new File(fileName) );    //创建文件扫描器
27                while (sc.hasNext()) {           //检查扫描器中是否还有可输入的数据
28                    int x = sc.nextInt();        //读取下一个 int 型整数
29                    double y = sc.nextDouble(); //读取下一个 double 型实数
30                    String str = sc.next();      //读取下一个字符串
31                    System.out.println( x +", " + y +", " + str );     //显示输入结果
32                }
33                sc.close();                      //关闭文件扫描器
34                System.out.println(fileName +"输入成功.");
35            }
36            catch(IOException e)                //处理 IOException 异常(勾选异常)
37            {  System.out.println( e.getMessage() );   }
38  }  }
```

在 Eclipse 集成开发环境中运行例 7-7 的程序,运行结果如图 7-9 所示。



图 7-9 例 7-7 程序的运行结果

例 7-7 中代码第 10 行直接使用类 PrintWriter 创建了一个字符型文件打印流类对象。

```
PrintWriter pw = new PrintWriter( fileName );  //创建字符型文件打印流
```

可以将上述代码改为如下形式:

```
PrintWriter pw = new PrintWriter( new BufferedWriter( new FileWriter(fileName) ));
```

这行代码的含义是:先将文件输出流类 FileWriter 的对象包装成带缓冲区的 BufferedWriter 类对象,这样就能利用内存缓冲区提高文件写入的速度;然后再将带缓冲区的 BufferedWriter 类对象包装成具有格式化输出功能的 PrintWriter 类对象。这是一种多级包装的形式(见图 7-10)。每增加一级包装,就得到一个功能更强的对象。多级包装在 Java 语言中被广泛使用。

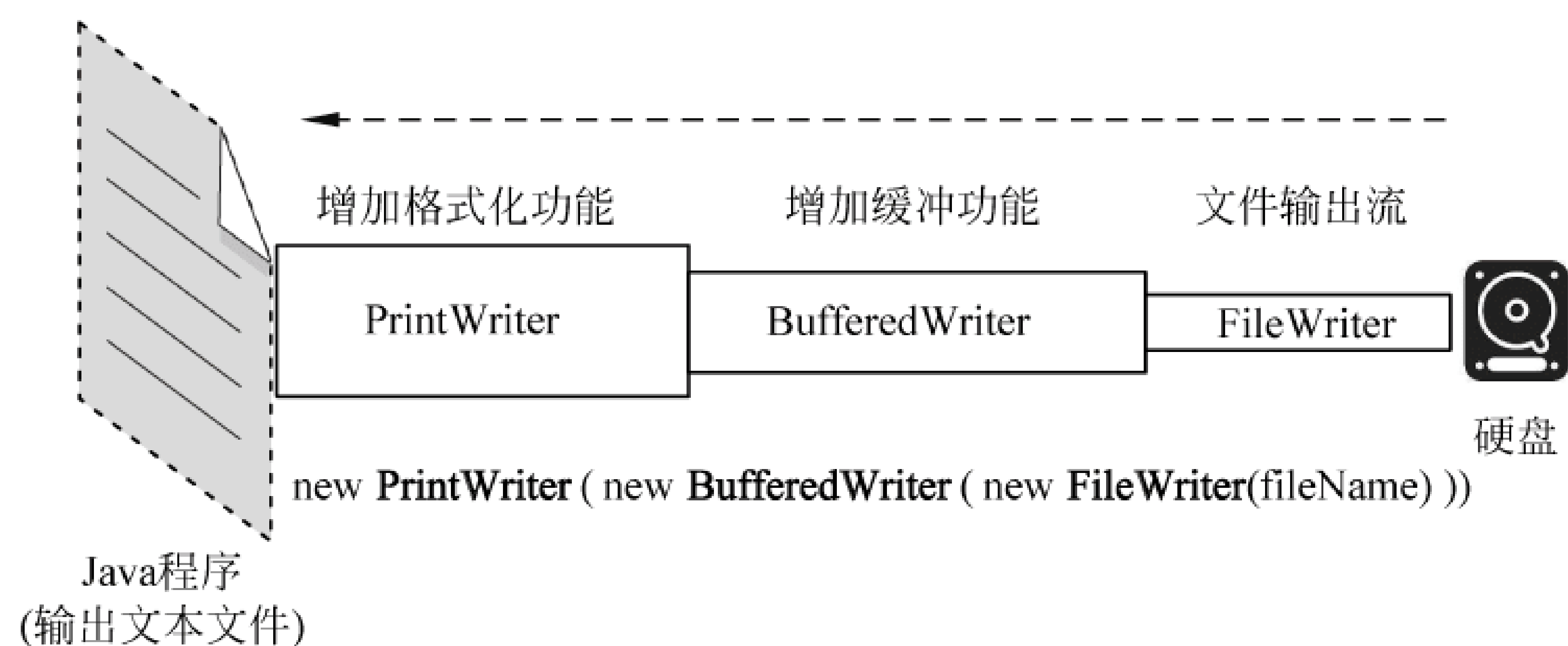


图 7-10 多级包装

本节习题

1. 文本文件一般不会存储字符(　　)。
   A. 'J'　　B. 'j'　　C. '8'　　D. Esc 键
2. 文本文件的扩展名一般是(　　)。
   A. .txt　　B. .doc　　C. .docx　　D. .jpg
3. 在文本文件中存储一个 int 型整数需要(　　)字节。
   A. 1　　B. 4　　C. 8　　D. 不确定
4. 在二进制文件中存储一个 int 型整数需要(　　)字节。
   A. 1　　B. 4　　C. 8　　D. 不确定
5. 文件类 File 中对文件或目录进行重命名的方法是(　　)。
   A. length()　　B. delete()
   C. renameTo()　　D. isDirectory()
6. 字符型文件输入流类 FileReader 直接继承了(　　)类。
   A. InputStream　　B. Reader
   C. InputStreamReader　　D. Scanner
7. 带缓冲区的字符型输入流类 BufferedReader 继承字符型输入流类 Reader，然后扩展了(　　)方法。
   A. ready()　　B. read()　　C. close()　　D. readLine()
8. 类(　　)具有将不同类型变量中的数据格式化输出到文本文件的功能。
   A. OutputStream　　B. Writer　　C. FileWriter　　D. PrintWriter

7.4 序列化及二进制文件 I/O

格式化是将不同类型的数据格式化成字符流。很多时候，程序也需要将所处理变量或对象中的数据**序列化**(serialization)成一个字节流。序列化的目的有两个。

(1) **将数据保存到外存文件**。通过序列化，可以将内存变量或对象中的数据序列化成

字节流，然后保存到外存文件中去，这被称为是数据的**持久化**(persistence)。保存在外存文件中的数据，即使在程序执行结束退出后也可以长期保存。再次执行程序时，可以将外存文件中的数据迅速恢复到内存变量或对象中，这被称为是数据的**反序列化**(deserialization)。

(2) **通过网络传输数据**。序列化成字节流之后的数据也可以通过网络传输给其他计算机。对方在接收到字节流之后，可以通过反序列化恢复出数据。

将数据序列化成字节流而不是格式化成字符流，这是因为序列化的处理速度相对较快，所得到的字节流数据量也更小。

将序列化后的字节流数据保存到外存文件，必须使用二进制文件格式。本节讲解序列化及二进制文件 I/O。通过网络传输序列化数据这部分的内容将在第 9 章讲解。

7.4.1 字节型输入输出流类族

为了对字节流数据进行输入输出，Java API 专门提供了两个字节型输入输出流类族(见图 7-11)。

- 字节型**输入流**类族，提供 byte 型数据的输入功能，根类是 **InputStream**。
- 字节型**输出流**类族，提供 byte 型数据的输出功能，根类是 **OutputStream**。

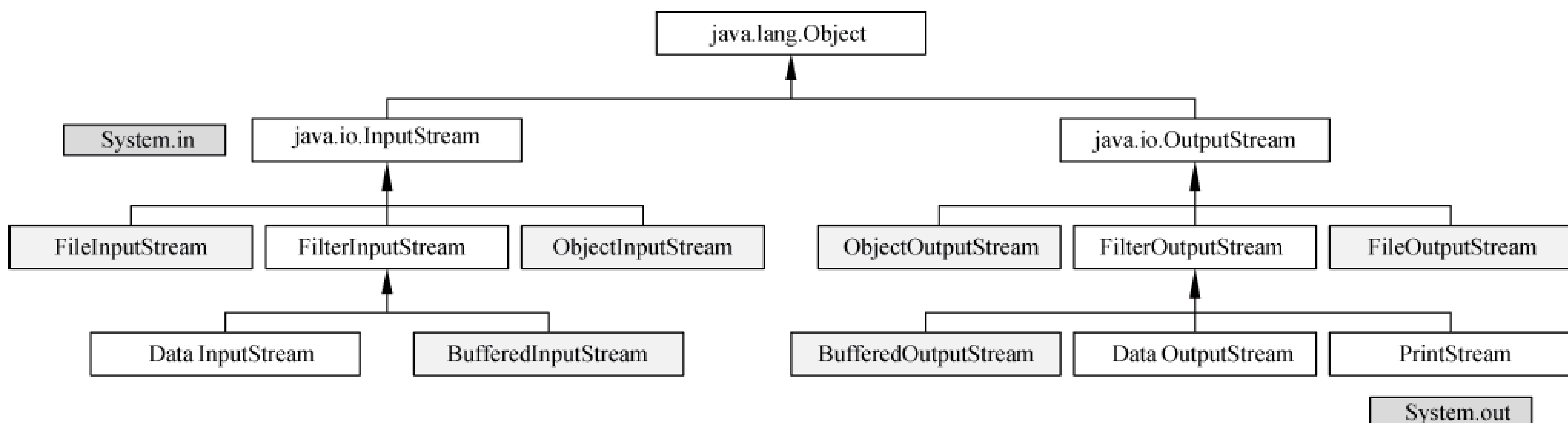


图 7-11 两个字节型输入输出流类族(未包含包名的类都属于 java.io 包)

图 7-11 说明如下。

(1) **二进制文件 I/O 类**。FileInputStream 是字节型文件输入流类，FileOutputStream 是字节型文件输出流类。使用这两个类即可实现二进制文件的输入输出。

(2) **带缓冲区的字节型输入输出流类**。BufferedInputStream 是带缓冲区的字节型输入流类，BufferedOutputStream 是带缓冲区的字节型输出流类。这是两个包装类，可以为二进制文件 I/O 类的对象增加缓冲区功能，这样可以提高二进制文件的读写速度。

(3) **带反序列化/序列化功能的字节型输入输出流类**。ObjectInputStream 是带反序列化功能的字节型输入流类，称为对象输入流类；ObjectOutputStream 是带序列化功能的字节型输出流类，称为对象输出流类。这是两个包装类，可以为二进制文件 I/O 类的对象增加反序列化或序列化功能。

7.4.2 简单数据的序列化及二进制文件 I/O

Java 语言有 8 种基本数据类型，例如 int 型或 double 型等。另外，字符串类 String 也是一种常用数据类型。对这 9 种数据类型的序列化比较简单，Java API 使用对象输出流类

ObjectOutputStream 为它们提供了序列化及输出方法，再使用对象输入流类 **ObjectInputStream** 为它们提供了输入及反序列化方法。这是两个包装类，可以通过包装二进制文件 **I/O** 类的对象实现完整的数据序列化及二进制文件 **I/O** 功能。

例 7-8 给出了一个简单数据序列化及二进制文件 I/O 的 Java 演示程序。

例 7-8 一个简单数据序列化及二进制文件 I/O 的 Java 演示程序(JObjectIO.java)

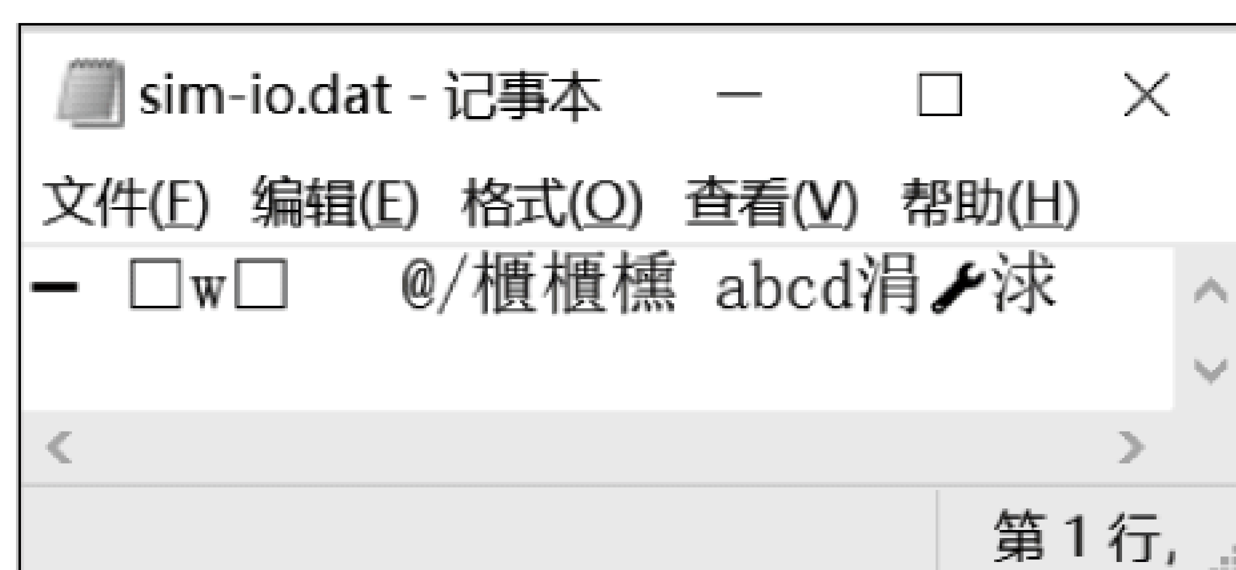
```
1  import java.io.*;                            //导入 java.io 包中的类
2  public class JObjectIO {                     //测试类
3      public static void main(String[] args) { //主方法
4          fwrite("d:/sim-io.dat");         //调用下面的方法 fwrite()输出一个二进制文件
5          fread("d:/sim-io.dat");  //调用下面的方法 fread()输入并显示二进制文件的内容
6      }
7      static  void fwrite(String fileName) {   //序列化并输出二进制文件的方法
8          try {  //必须处理勾选异常 IOException
9              //先创建字节型文件输出流对象,再包装成带序列化功能的输出流对象
10             FileOutputStream fos = new FileOutputStream(fileName);
11             ObjectOutputStream oos = new ObjectOutputStream( fos );
12             int x = 10;  double y = 15.8;  String str = "abcd 中国";  //简单数据
13             oos.writeInt(x);                 //序列化并输出一个 int 型整数(4 字节)
14             oos.writeDouble(y);              //序列化并输出一个 double 型实数(8 字节)
15             oos.writeUTF(str);  //序列化并输出一个字符串(UTF-16 被转换为 UTF-8)
16             oos.close();                     //关闭输出流
17             System.out.println(fileName +"输出成功.");  System.out.println();
18         }
19         catch(IOException e)                 //处理 IOException 异常(勾选异常)
20         {  System.out.println( e.getMessage() );  }
21     }
22     static void fread(String fileName) {     //输入二进制文件并反序列化的方法
23         try {  //必须处理勾选异常 IOException
24             //先创建字节型文件输入流对象,再包装成带反序列化功能的输入流对象
25             FileInputStream fis = new FileInputStream(fileName);
26             ObjectInputStream ois = new ObjectInputStream( fis );
27             int x = ois.readInt();           //输入并反序列化一个 int 型整数(4 字节)
28             double y = ois.readDouble();  //输入并反序列化一个 double 型实数(8 字节)
29             String str = ois.readUTF();  //输入并反序列化字符串(UTF-8 转换为 UTF-16)
30             System.out.println( x +", " +y +", " +str );     //显示输入结果
31             ois.close();                     //关闭数据输入流
32             System.out.println(fileName +"输入成功.");
33         }
34         catch(IOException e)                 //处理 IOException 异常(勾选异常)
35         {  System.out.println( e.getMessage() );  }
36 }  }
```

在 Eclipse 集成开发环境中运行例 7-8 的程序，运行结果如图 7-12(a)所示。检查图 7-12(a)中经序列化和反序列化后所显示出的数据。可以看出，不同类型数据可经序列化输出到二进制文件，然后再通过反序列化准确还原回内存变量中。

如果使用其他程序，例如 Windows 的"记事本"程序，打开例 7-8 所创建的数据文件 sim-io.dat，会发现所显示的内容是乱码，如图 7-12(b)所示。



(a) 例7-8程序的运行结果



(b) 使用记事本程序打开二进制文件sim-io.dat会显示乱码

图 7-12 简单数据序列化及二进制文件 I/O

序列化及二进制文件 I/O 的编程要点如下。

(1) **对象输出流类** ObjectOutputStream。对象输出流类 ObjectOutputStream 可以将字节型文件输出流类 FileOutputStream 对象包装成一个具有序列化功能的新对象，这样就能对不同类型的数据进行序列化并输出到二进制文件中去。程序员还可以使用类 BufferedOutputStream 实现带缓冲区的二进制文件输出功能。例如：

```
FileOutputStream     fos = new FileOutputStream(fileName);//先创建字节型文件输出流对象
BufferedOutputStream bos = new BufferedOutputStream(fos);  //包装成带缓冲区的输出流对象
ObjectOutputStream   oos = new ObjectOutputStream(bos);     //再包装成带序列化的输出流对象
```

(2) **对象输入流类** ObjectInputStream。对象输入流类 ObjectInputStream 可以将字节型文件输入流类 FileInputStream 对象包装成一个具有反序列化功能的新对象，这样就能从二进制文件中输入数据并进行反序列化。程序员还可以使用类 BufferedInputStream 实现带缓冲区的二进制文件输入功能。例如：

```
FileInputStream     fis = new FileInputStream(fileName); //先创建字节型文件输入流对象
BufferedInputStream bis = new BufferedInputStream(fis);  //包装成带缓冲区的输入流对象
ObjectInputStream   ois = new ObjectInputStream(bis);     //再包装成带反序列化的输入流对象
```

(3) **基本数据类型数据的序列化**。序列化基本数据类型的数据，就是直接将其内存的存储形式看作字节流，不做任何格式转换。例如，内存中的 int 型整数可以直接看作是一个 4 字节的字节流，而 double 型实数则是一个 8 字节的字节流。

(4) **字符串数据的序列化**。在将 String 类型字符串序列化成字节流时,方法 writeUTF()会将先将内存中的 UTF-16 编码转换成 UTF-8 编码(与标准 UTF-8 略有不同),然后再序列化成字节流。而在反序列化 UTF-8 编码的字节流时,方法 readUTF()也会将 UTF-8 编码转回 UTF-16 编码。UTF-16 是 Java 语言内部的字符编码格式。

7.4.3 对象序列化

很多时候,程序也需要将类类型的对象数据序列化成字节流,然后保存到二进制文件中,或通过网络进行传输。在 Java 语言中,只有实现"可序列化"接口 Serializable 的类才能被序列化。接口 Serializable 的定义代码如下:

```
public interface Serializable;                              //接口 Serializable 的定义代码
```

接口 Serializable 是一个标记接口,即不包含任何成员,是一个空接口。实现 Serializable 接口的目的是为类激活(或称启用)序列化功能。

对象输出流类 ObjectOutputStream 提供了方法成员 **writeObject**()用于序列化并输出对象数据,而对象输入流类 ObjectInputStream 则提供了方法成员 **readObject**()用于输入并反序列化对象。例 7-9 给出了一个对钟表类 Clock 对象进行序列化和反序列化的 Java 演示程序。

例 7-9 对钟表类 Clock 对象进行序列化和反序列化的 Java 演示程序(JObjectIO.java)

```
1  import java.io.*;                                  //导入 java.io 包中的类
2  public class JObjectIO {                           //测试类
3      public static void main(String[] args) {       //主方法
4          fwrite("d:/obj-io.dat");               //调用下面的方法 fwrite()输出一个二进制文件
5          fread("d:/obj-io.dat");  //调用下面的方法 fread()输入并显示二进制文件的内容
6      }
7      static  void fwrite(String fileName) {         //序列化并输出二进制文件的方法
8          try {  //必须处理勾选异常 IOException
9                //先创建字节型文件输出流对象,再包装成带序列化功能的输出流对象
10               FileOutputStream fos = new FileOutputStream(fileName);
11               ObjectOutputStream oos = new ObjectOutputStream( fos );
12               Clock c1 = new Clock(8, 30, 15);  //创建两个将被序列化的钟表对象
13               Clock c2 = new Clock(10, 30, 15);
14               oos.writeObject(c1);  oos.writeObject(c2);  //序列化并输出对象
15               oos.close();                          //关闭输出流
16               System.out.println(fileName + "输出成功.");  System.out.println();
17           }
18           catch(IOException e)                      //处理 IOException 异常(勾选异常)
19           {  System.out.println( e.getMessage() );  }
20     }
```

```
21      static void fread(String fileName) {           //输入二进制文件并反序列化的方法
22          try {  //必须处理勾选异常 IOException
23              //先创建字节型文件输入流对象,再包装成带反序列化功能的输入流对象
24              FileInputStream fis = new FileInputStream(fileName);
25              ObjectInputStream ois = new ObjectInputStream( fis );
26              Clock c1 = (Clock)ois.readObject(); //输入并反序列化第 1 个钟表对象
27              Clock c2 = (Clock)ois.readObject(); //输入并反序列化第 2 个钟表对象
28              c1.show();  c2.show();              //显示反序列化得到的钟表对象时间
29              ois.close();                        //关闭数据输入流
30              System.out.println(fileName + "输入成功.");
31          }
32          catch(IOException e)                 //处理 IOException 异常(勾选异常)
33          {  System.out.println( e.getMessage() );  }
34          catch(ClassNotFoundException e)      //处理勾选异常 ClassNotFoundException
35          {  System.out.println( e.getMessage() );  }
36  }  }
37
38  class Clock implements Serializable {          //定义钟表类时激活序列化功能
39      //必须添加一个 long 型字段 serialVersionUID,为类指定一个序列化编号
40      private static final long serialVersionUID = 2018L;  //本例将编号指定为 2018
41      private int hour, minute, second;          //字段: 时、分、秒
42      public  void show()                        //显示时间
43      {  System.out.println( hour + ":" + minute + ":" + second );  }
44      public Clock(int h, int m, int s)          //构造方法
45      {  hour = h;  minute = m;  second = s;  }
46  }
```

在 Eclipse 集成开发环境中运行例 7-9 的程序,运行结果如图 7-13 所示。检查图中经序列化和反序列化后所显示出的钟表对象的时间。可以看出,钟表对象可经序列化输出到二进制文件,然后再通过反序列化准确还原回内存对象中。

对象序列化,指的是将对象中字段成员所保存的数据序列化成字节流。如果某些字段成员不希望被序列化(例如保存银行账号或密码的字段),则应当在定义类时为这些字段添加修饰符 **transient**,将它们定义成"非持久化"字段。Java 语言中,类的非持久化字段和静态字段都不参与今后的序列化或反序列化处理。



图 7-13 例 7-9 程序的运行结果

7.4.4 对象输入输出流类说明文档

请读者阅读下面的对象输入流类 ObjectInputStream 和对象输出流类 ObjectOutputStream 说明文档。

java. io. **ObjectInputStream** 类说明文档

public class **ObjectInputStream**
extends **InputStream**
implements **ObjectInput**，**ObjectStreamConstants**

	修　饰　符	类成员(节选)	功 能 说 明
1		**ObjectInputStream**(InputStream in)	构造方法
2		int **read**(byte[] b，int off，int len)	按字节流读出 len 个字节
3		byte **readByte**()	读出一个 byte 型整数(1 字节)
4		int **readUnsignedByte**()	读出一个无符号单字节整数(1 字节)
5		int **readInt**()	读出一个 int 型整数(4 字节)
6		float **readFloat**()	读出一个 float 型实数(4 字节)
7		double **readDouble**()	读出一个 double 型实数(8 字节)
8		char **readChar**()	读出一个 char 型字符(2 字节)
9		String **readUTF**()	读出 UTF-8 编码的字符串
10		Object **readObject**()	读出一个对象然后反序列化
11		void **close**()	关闭对象输入流

…

java. io. **ObjectOutputStream** 类说明文档

public class **ObjectOutputStream**
extends **OutputStream**
implements **ObjectOutput**，**ObjectStreamConstants**

	修　饰　符	类成员(节选)	功 能 说 明
1		**ObjectOutputStream**(OutputStream out)	构造方法
2		void **write**(byte[] b，int off，int len)	按字节流写入 len 个字节
3		void **writeByte**(int v)	写入一个字节(v 的低字节)
4		void **writeInt**(int v)	写入一个 int 型整数(4 字节)
5		void **writeFloat**(float v)	写入一个 float 型实数(4 字节)
6		void **writeDouble**(double v)	写入一个 double 型实数(8 字节)
7		void **writeChar**(int v)	写入一个字符(v 的 2 个低字节)
8		void **writeUTF**(String str)	转成 UTF-8 编码字节流后再写入
9		void **writeObject**(Object obj)	序列化对象然后写入其字节流
10		void **flush**()	立即输出缓存里的内容
11		void **close**()	关闭对象输出流

…

本节习题

1. 下列关于序列化的描述中，错误的是(　　)。
   A. 通过序列化，可以将内存变量或对象中的数据序列化成字节流
   B. 序列化成字节流之后的数据可以保存到二进制文件中
   C. 序列化成字节流之后的数据可以保存到文本文件中

D. 序列化成字节流之后的数据可以通过网络进行传输

2. 字节型文件输入流类 FileInputStream 直接继承了(　　)类。

A. InputStream　　B. Reader

C. InputStreamReader　　D. Scanner

3. 对象输出流类 ObjectOutputStream 中将 int 型整数序列化并输出的方法是(　　)。

A. writeUTF()　　B. writeInt()

C. writeDouble()　　D. writeObject()

4. 对象输出流类 ObjectOutputStream 中将字符串序列化并输出的方法是(　　)。

A. writeUTF()　　B. writeInt()

C. writeDouble()　　D. writeObject()

5. 对象输出流类 ObjectOutputStream 中将对象数据序列化并输出的方法是(　　)。

A. writeUTF()　　B. writeInt()

C. writeDouble()　　D. writeObject()

6. 对象输入流类 ObjectInputStream 中输入并反序列化 int 型整数的方法是(　　)。

A. readUTF()　　B. readInt()

C. readDouble()　　D. readObject()

7. 对象输入流类 ObjectInputStream 中输入并反序列化字符串的方法是(　　)。

A. readUTF()　　B. readInt()

C. readDouble()　　D. readObject()

8. 如果一个类希望通过 Java API 的对象输入输出流类进行序列化输入输出，则这个类必须实现(　　)接口。

A. Serializable　　B. Comparable

C. Cloneable　　D. Map

7.5 文本处理

本节讲解文本的编辑和处理。**文本编辑**通常使用图形用户界面。用户在图形界面的文本编辑框(例如 Java API 的文本区域类 JTextArea)中编辑文本，然后将编辑好的内容保存到文本文件中。**文本处理**通常包括分词、查找、替换等字符串操作，本节将介绍文本处理中一项非常重要的技术——**正则表达式**(regular expression)。

7.5.1 文本编辑

图 7-14 给出一个使用 Java API 编写的简单的文本编辑器演示程序，其功能描述如下。

- 使用图形界面，方便用户操作。
- 在图形界面中使用多行文本编辑框(JTextArea)来输入、编辑文字。
- 添加“保存”按钮(JButton)，其功能是将多



图 7-14　一个简单的文本编辑器演示程序

行文本编辑框中的内容保存到文本文件。

- 添加“打开”按钮(JButton),其功能是将文本文件中的内容读入多行文本编辑框。
- 打开或保存文件时使用文件选择对话框(JFileChooser)来选择文件。

例 7-10 给出了实现图 7-14 文本编辑器功能的完整 Java 示例代码。

例 7-10 一个使用 Java API 编写的简单的文本编辑器演示程序(JNotepad.java)

```
1  import java.awt.*;                                  //导入 java.awt 包中的类
2  import java.awt.event.*;                            //导入 java.awt.event 包中的事件类
3  import javax.swing.*;                               //导入 javax.swing 包中的图形组件类
4  import java.io.*;                                   //导入 java.io 包中的输入输出流类
5
6  public class JNotepad {                             //测试类
7      public static void main(String[] args) {        //主方法
8          MainWnd w = new MainWnd();                  //创建并显示主窗口对象
9  } }
10
11 class MainWnd  extends JFrame {                     //扩展 JFrame
12     JTextArea text = new JTextArea(10, 20);         //添加一个多行文本编辑框
13     JButton bOpen = new JButton( "打开" );           //打开文件按钮
14     JButton bSave = new JButton( "保存" );           //保存文件按钮
15     public MainWnd() {                              //构造方法
16         setTitle( "文本编辑器演示程序" );              //初始化窗口
17         setSize(450, 200);  setLocation(100, 100);  setVisible(true);
18         setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
19         text.setBackground(Color.WHITE);
20         //布局多行文本编辑框和按钮组件
21         JPanel bPane = new JPanel();                //创建一个按钮面板(默认流式布局)
22         bPane.add(bOpen);  bPane.add (bSave); //将两个按钮放入按钮面板
23         Container cp = getContentPane ();           //获得窗口的内容面板(默认边框布局)
24         cp.add(bPane, BorderLayout.NORTH);  cp.add (text, BorderLayout.CENTER);
25         cp.validate();                              //检查并自动布局内容面板里的组件
26         //为"保存"按钮添加监听器,保存文本文件
27         bSave.addActionListener( new ActionListener() {      //匿名类
28             public void actionPerformed (ActionEvent e) {
29                 JFileChooser fc = new JFileChooser("d:/");//创建文件选择对话框
30                 fc.showSaveDialog (null);                   //弹出保存对话框
31                 File f = fc.getSelectedFile();              //获得所选择的文件
32                 try {  //必须处理异常 IOException(勾选异常)
33                     //创建文本文件输出流,然后包装成带缓冲区的输出流
34                     BufferedWriter out = new BufferedWriter ( new FileWriter(f) );
35                     String txt = text.getText();  //取出多行文本编辑框里的内容
36                     int p1 = 0, p2;
37                     while (true) {          //处理字符串中的换行符'\n'
38                         p2 = txt.indexOf('\n', p1);
39                         if (p2 == -1)   //最后一行文本,输出后结束处理
40                         {  out.write( txt.substring(p1) );  break;  }
41                         else {          //取出一行文本,输出后添加换行符
42                             out.write( txt.substring(p1, p2) );  out.newLine();
```

```
43                              p1 = p2 + 1; //继续处理下一行
44                          }   }
45                          out.close();          //关闭文本文件输出流
46                      }
47                      catch(IOException eIO)    //捕获并处理 IOException 异常(勾选异常)
48                      {  System.out.println( eIO.getMessage() );  }
49              }} );
50              //为"打开"按钮添加监听器,打开已有的文本文件
51              bOpen.addActionListener( new ActionListener() {    //匿名类
52                  public void actionPerformed(ActionEvent e) {
53                      JFileChooser fc = new JFileChooser("d:/");//创建文件选择对话框
54                      fc.showOpenDialog(null);                   //弹出打开对话框
55                      File f = fc.getSelectedFile();             //获得所选择的文件
56                      try {  //必须处理异常 IOException(勾选异常)
57                          //创建文本文件输入流,然后包装成带缓冲区的输入流
58                          BufferedReader in = new BufferedReader( new FileReader(f) );
59                          text.setText("");                      //清空多行文本编辑框
60                          String s;
61                          while ((s = in.readLine()) != null)//读出文本文件中的一行
62                          {  text.append(s + "\n");  }           //添加到多行文本编辑框里
63                          in.close();                            //关闭文本文件输入流
64                      }
65                      catch(IOException eIO)      //捕获并处理 IOException 异常(勾选异常)
66                      {  System.out.println( eIO.getMessage() );}
67              }} );
68  }  }
```

请读者仔细阅读例 7-10 的程序代码,这样可以进一步加深对之前所学习的图形用户界面程序和输入输出流知识的理解。

例 7-10 实现的是一个文本编辑器程序,可以将输入到内存字符串中的信息保存成外存文本文件;也可以将外存文本文件里的内容读入内存字符串,然后继续进行处理。文本文件存储的文本内容可以是一篇文章。例如:

> This lesson covers the Java platform classes used for basic IO. It first focuses on IO Streams, a powerful concept that greatly simplifies IO operations. The lesson also looks at serialization, which lets a program write whole objects out to streams and read them back again.

文章由段落组成,段落包含句子,句子又包含单词。除了常规编辑,计算机程序还可以对文章中的文本内容进行更高级的分析或处理,例如分词、查找、替换等。

图 7-14 演示的是在文本编辑器中输入一份通讯录。其中的每一行都是一条记录,每条记录包含多个由逗号分隔的数据项(或称为字段)。以换行分隔记录,以逗号分隔数据项,这种文本格式称为**逗号分隔数据**(Comma-Separated Values, **CSV**)格式。采用 CSV 格式存储数据的文本文件被称为 CSV 文件,扩展名一般为**.csv**。CSV 文件通常用作数据备份或在不同程序间交换数据。处理 CSV 文件,需要对每一行文本字符串进行分词,提取出其中的各个数据项,然后再进行处理。

在文本文件基础上制定不同的内容格式,这样可以设计出不同用途的文本文件。例如,CSV 文件就是一种以文本形式存储表格数据的文本文件,HTML 文件则是一种以文本形

式存储网页数据的文本文件(扩展名通常为. **html** 或. **htm**)。

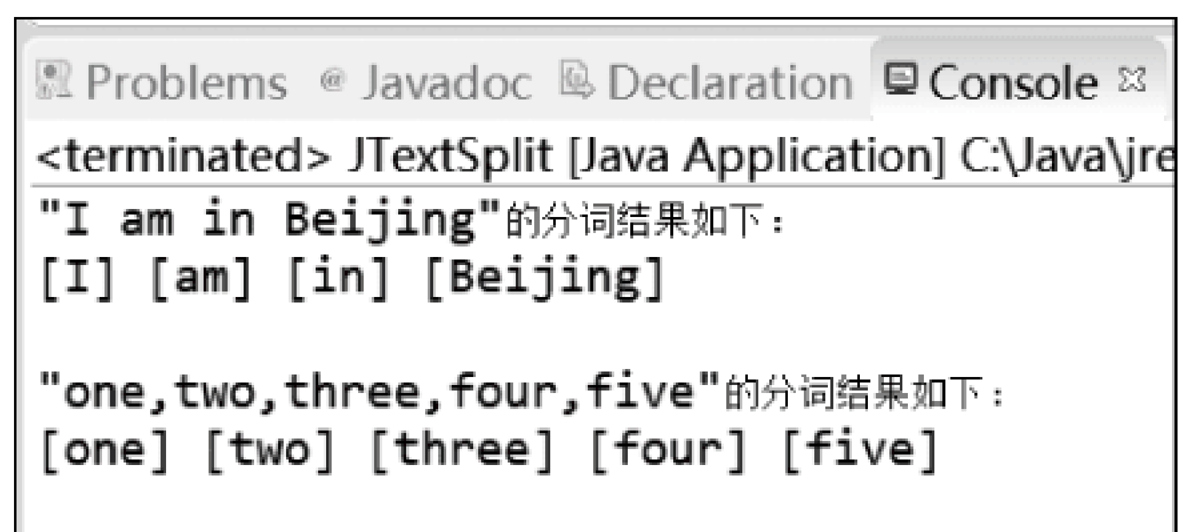
7.5.2 文本分词

对字符串里的内容进行**分词**(word segmentation)是很多后续高级文本处理技术的基础。Java API 中的字符串类 String 就为程序员提供了一个分词方法 **split**()。例 7-11 给出了一个使用字符串类 String 进行分词的 Java 演示程序。

例 7-11 一个使用字符串类 String 进行分词的 Java 演示程序(JTextSplit. java)

```
1  public class JTextSplit {                                    //测试类
2      public static void main(String[ ] args) {               //主方法
3          String str = "I am in Beijing";                      //待分词的字符串
4          String words[ ];                                     //字符串数组,用于保存分词结果
5          //对存储在 str 中的字符串"I am in Beijing"进行分词
6          words = str.split(" ");                              //将空格作为分隔符进行分词
7          System.out.println("\"" + str + "\"的分词结果如下: " );
8          for (String w: words)                                //显示分词结果
9          {   System.out.print("[" + w + "] ");   }
10         System.out.println();  System.out.println();         //换行
11         //再对字符串"one,two,three,four,five"进行分词
12         str = "one,two,three,four,five";                     //待分词的字符串
13         words = str.split(",");                              //将逗号作为分隔符进行分词
14         System.out.println("\"" + str + "\"的分词结果如下: " );
15         for (String w: words)                                //显示分词结果
16         {   System.out.print("[" + w + "] ");   }
17         System.out.println();
18 }  }
```

在 Eclipse 集成开发环境中运行例 7-11 的程序,运行结果如图 7-15 所示。


Problems Javadoc Declaration Console
<terminated> JTextSplit [Java Application] C:\Java\jre
"I am in Beijing"的分词结果如下:
[I] [am] [in] [Beijing]

"one,two,three,four,five"的分词结果如下:
[one] [two] [three] [four] [five]

图 7-15 例 7-11 程序的运行结果

图 7-15 所示两个字符串的分隔符比较规范,或都是空格,或都是逗号。使用" "或","的形式就能描述这样比较简单分隔符。如果分隔符比较复杂,例如:

"one,two three,four five"

这个字符串所使用的分隔符不是很规范,有的用空格,有的用逗号。该如何描述这样比较复杂的分隔符呢? 这时就需要用到文本处理中一项非常重要的技术——正则表达式。

7.5.3 正则表达式

一个字符序列可以包含哪些字符,各字符之间又有什么样的排列规律,这被称为字符序列的**模式**(pattern)。计算机语言使用**正则表达式**来描述字符序列的模式。正则表达式具有专门的**语法规则**,可以描述各种复杂的字符序列模式。

应用正则表达式,就是先按照语法编写描述某种字符序列模式的正则表达式,然后用该

正则表达式去**匹配**(match)字符串,检查字符串是否符合规定的模式,或是在字符串中查找符合规定模式的子字符串。

为便于阅读理解,本节以表格的形式对常用正则表达式语法进行了汇总、分类,其中包括模式说明、正则表达式语法,另外还给出了与正则表达式相匹配的正例以及不匹配的反例。下面按照从易到难的顺序给出4张表(见表7-1～表7-4),分别讲解如何使用正则表达式来描述**单个字符**、**单词**(多个字符)、**词组**(多个单词)以及**句子**(或称整行)的模式。

表7-1　描述单个字符模式的正则表达式

模式说明(单个字符模式)	正则表达式语法	正例与反例
某个特定的字符。例如字母a	"a"	正例:"a" 反例:"b"、"A"、"1"、","…
任意字符(除了换行、回车)。注:"."被称为通配符	"."	正例:"a"、"A"、"1"、","… 反例:"\n"、"\r"
某几个特定字符中的一个。例如,字母a、b、d中的一个	"[abd]"	正例:"a"、"b"、"d" 反例:"c"、"e"、"1"、","…
某几个连续字符中的一个。例如,字母a～d中的一个	"[a-d]"	正例:"a"、"b"、"c"、"d" 反例:"e"、"A"、"1"、","…
除去几个特定字符之外的其他字符。例如,除去字母a、b、d之外的其他字符	"[^abd]"	正例:"c"、"e"、"1"、","… 反例:"a"、"b"、"d"
除去某几个连续字符之外的其他字符。例如,除去字母a～d之外的其他字符	"[^a-d]"	正例:"e"、"A"、"1"、","… 反例:"a"、"b"、"c"、"d"
数字字符,即0～9的字符	"[0-9]"或简写成"\d"	正例:"0"、"1"、"2"、"3"… 反例:"a"、"b"、"A"、","…
非数字字符	"[^0-9]"或简写成"\D"	正、反例与上一格相反
空白字符,即空格、Tab键、换行等	"[\n\r\t\x0B\f]" 或简写成"\s"	正例:" "、"\n"、"\r"、"\t"… 反例:"a"、"A"、"1"、","…
非空白字符	"[^\s]"或简写成"\S"	正、反例与上一格相反
单词字符,即字母或数字字符	"[a-zA-Z_0-9]" 或简写成"\w"	正例:"a"、"z"、"A"、"1"… 反例:""、","、"_"、"$"…
非单词字符	"[^\w]"或简写成"\W"	正、反例与上一格相反

表7-2　描述单词(多个字符)模式的正则表达式

模式说明(单词模式)	正则表达式语法	正例与反例
某个特定的单词。例如abc	"abc"	正例:"abc" 反例:"a"、"ab"、"Abc"、"ab2"…
任意多个一样的字符,不能是0个。例如3个字母a,即aaa	"a+"	正例:"a"、"aa"、"aaa"… 反例:""、"b"、"Ab"…
任意多个一样的字符,可以是0个。例如3个字母a,即aaa	"a*"	正例:""、"a"、"aa"、"aaa"… 反例:"b"、"Ab"…
某个可以省略的字符。例如,字母a可以省略	"a?"	正例:""、"a" 反例:"b"、"A"、"Ab"…
n个一样的字符。例如3个字母a,即aaa	"a{3}"	正例:"aaa" 反例:"a"、"aa"、"Aaa"…

续表

模式说明(单词模式)	正则表达式语法	正例与反例
至少 n 个一样的字符。例如,至少 3 个字母 a	"a{3,}"	正例:"aaa"、"aaaa"、"aaaaa"… 反例:"a"、"aa"、"Aaa"…
m～n 个一样的字符。例如,1～3 个字母 a	"a{1, 3}"	正例:"a"、"aa"、"aaa" 反例:"aaaa"、"Aa"、"Aaa"…
含有某个或某几个字符的单词。例如 a0、a1、a2	"a[\d]"	正例:"a0"、"a1"、"a2"… 反例:"a"、"ab"、"b1"…
带可重复字符的单词。例如 abc、abbc、ac,其中的字母 b 可重复	"ab * c"	正例:"ac"、"abc"、"abbc"… 反例:"a"、"c"、"acb"…

表 7-3 描述词组(多个单词)模式的正则表达式

模式说明(词组模式)	正则表达式语法	正例与反例
某个特定的词组。例如,某个文件目录 c:/java/src 可认为是由 3 个单词组成的词组。正则表达式使用小括号"()"进行分组	"c:/java/src"或定义成词组:"(c:)(/java)(/src)" **注:** 3 个单词的序号依次为 0、1、2	正例:"c:/java/src" 反例:"c:/java/bin"、"d:/java/src"…
含有某个或某几个特定字符的词组。例如,硬盘分区 c 或 d 上的目录/java/src	"([cd]:)(/java)(/src)"	正例:"c:/java/src"、"d:/java/src"… 反例:"c:/java/bin"、"e:/java/src"…
用"|"指定含有某几个特定单词的词组。例如,指定目录 c:/java 下的/src 或/bin 子目录	"(c:)(/java)/(src|bin)"	正例:"c:/java/src"、"c:/java/bin" 反例:"c:/java/doc"、"c:/java/"…
用"+"表示含有重复字符的词组。例如,目录 c:/java 下的所有子目录。**注:** "."是通配符	"(c:)(/java)(/.+)"	正例:"c:/java/src"、"c:/java/bin"… 反例:"c:/cpp/src"、"c:/cpp/hpp"…
用"+"表示含有重复单词的词组。例如,目录 c:/java/java/src	"(c:)(/java)+(/src)"	正例:"c:/java/src"、"c:/java/java/src"… 反例:"c:/cpp/src"、"c:/cpp/hpp"…

表 7-4 描述句子(或称整行)模式的正则表达式

模式说明(句子模式)	正则表达式语法	正例与反例
用"^"指定以某个字符或单词开头的句子。例如,指定位于硬盘分区 c:上的文件目录/java 或/java/src	"^(c:)(/java)(/src)?"	正例:"c:/java"、"c:/java/src" 反例:"c:/; c:/java"、"d:/java"…
用"$"指定以某个字符或单词结尾的句子。例如,指定以 src 结尾的文件目录 c:/src 或 c:/java/src	"(c:)(/java)? (/src)$"	正例:"c:/src"、"c:/java/src" 反例:"c:/src/img"、"c:/src; c:/"…
用"^"和用"$"指定完整的句子(整行)。例如 c:/java/src	"^(c:)(/java)(/src)$"	正例:"c:/java/src" 反例:"c:/; c:/java/src"、"c:/java/src/img"…

续表

模式说明(句子模式)	正则表达式语法	正例与反例
用"\b"指定句子中某个单词前面或后面的字符必须是非单词字符,即不能是字母或数字字符	".* \bdog\b.* "	正例:"I love dog. " 反例:"I love doggie. "…
用"\B"指定句子中某个单词前面或后面的字符必须是单词字符,即只能是字母或数字字符	".* \bdog\B.* "	正例:"I love doggie. " 反例:"I love dog. "…

下面给出几个常用的正则表达式例子。

中国的邮政编码:"\b[1-9]\d{5}\b"

HTTP网址:"^http://www(.(\w+([- _]\w+)*)+$"

电子邮件地址:"\w+([- _]\w+)*@\w+([- _.]\w+)*.\w+"

如果在Java程序中以字符串常量形式书写上述正则表达式,则其中的反斜杠"\"需使用转义形式,将其写成"\\"。

7.5.4　模式类Pattern与匹配器类Matcher

Java API提供了一个模式类Pattern,用于保存描述字符序列模式的正则表达式。Java API还提供了一个使用正则表达式对文本字符串进行词法分析和处理的匹配器类Matcher,一个匹配器对象所保存的是词法分析和处理的结果,或称匹配结果。这两个类被定义在java. util. regex包中。

1. 模式类Pattern

模式类Pattern主要有以下3项功能。

(1) 模式类Pattern可以描述比较复杂的分隔符模式,用于对文本字符串进行分词。

(2) 模式类Pattern可用于检查文本字符串是否符合某种规定的格式。

(3) 通过模式类Pattern创建下一步词法分析所需的匹配器对象。

请读者阅读下面的模式类Pattern说明文档。

java. util. regex. **Pattern** 类说明文档

public final class **Pattern**
extends **Object**
implements **Serializable**

	修　饰　符	类成员(节选)	功 能 说 明
1	static	Pattern **compile**(String regex)	编译字符串形式的正则表达式,将其转换成模式对象的形式
2	static	Pattern **compile**(String regex, int flags)	编译字符串形式的正则表达式
3	static	boolean **matches**(String regex, CharSequence input)	检查字符串input是否符合正则表达式regex规定的模式

续表

	修　饰　符	类成员(节选)	功 能 说 明
4		String[] **split**(CharSequence input)	根据正则表达式指定的分隔符对字符串进行分词
5		String **pattern**()	返回描述正则表达式的字符串
6		Matcher **matcher**(CharSequence input)	使用正则表达式对字符串 input 进行处理,返回一个匹配器对象(将用于下一步词法分析或处理)
…			

例 7-12 给出了一个使用模式类 Pattern 进行分词的 Java 演示程序。

例 7-12　一个使用模式类 Pattern 进行分词的 Java 演示程序(JPatternTest.java)

```
1  import java.util.regex.*;              //导入 java.util.regex 包中与正则表达式相关的类
2  public class JPatternTest {                        //测试类
3      public static void main (String[] args) {      //主方法
4          String str = "one,two three, four     five"; //格式比较随意的字符串
5          Pattern p = Pattern.compile ("[, ]+"); //描述由任意多个逗号或空格组成的分隔符
6          String words[] = p.split (str);            //使用正则表达式进行分词
7          System.out.println("\"" + str + "\"的分词结果如下: ");
8          for (String s: words)                      //显示分词结果,用[ ]将每个单词括起来
9          {  System.out.print("[" + s + "]");  }
10         System.out.println();
11 } }
```

在 Eclipse 集成开发环境中运行例 7-12 的程序,其运行结果如下:

```
"one,two three, four     five"的分词结果如下:
[one][two][three][four][five]
```

例 7-13 给出了一个使用模式类 Pattern 检查 E-mail 邮箱格式的 Java 演示程序。

例 7-13　使用模式类 Pattern 检查 E-mail 邮箱格式的 Java 演示程序(JMailTest.java)

```
1  import java.util.regex.Pattern;                 //导入 java.util.regex 包中的类 Pattern
2  public class JMailTest {                        //测试类
3      public static void main(String[] args) {    //主方法
4          String mFormat = "^\\w+([-_]\\w+)*@\\w+([-_.]\\w+)*.\\w+$";
                                                   //E-mail 格式
5          String mail1 = "kan-daohong@cau.edu.cn";     //符合 E-mail 格式的邮箱
6          String mail2 = "kan-daohong.cau.edu.cn";     //不符合 E-mail 格式的邮箱
7          //下面调用模式类 Pattern 的静态方法 matches()检查 mail1 和 mail2 的邮箱格式
8          if (Pattern.matches(mFormat, mail1) == true)   //检查 mail1 是否符合邮箱格式
9              System.out.println(mail1 + ": 合法邮箱");
```

```
10          else System.out.println(mail1 + ": 非法邮箱");
11          if (Pattern.matches(mFormat, mail2) == true)  //检查 mail2 是否符合邮箱格式
12              System.out.println(mail2 + ": 合法邮箱");
13          else  System.out.println(mail2 + ": 非法邮箱");
14  } }
```

在 Eclipse 集成开发环境中运行例 7-13 的程序，其运行结果如下：

```
kan-daohong@cau.edu.cn: 合法邮箱
kan-daohong.cau.edu.cn: 非法邮箱
```

2. 匹配器类 Matcher

使用正则表达式对某个文本字符串进行词法分析或处理的过程可分为如下 3 步。

（1）创建描述正则表达式的模式对象。例如创建一个描述 E-mail 邮箱格式的模式对象 p：

```
Pattern p = Pattern.compile( "([^@]+)@([\\w]+)(\\.[\\w]+)+" );
```

（2）使用模式对象创建处理某个文本字符串的匹配器对象。例如，使用描述 E-mail 邮箱格式的模式对象 p 创建一个处理文本字符串 str 的匹配器对象 m：

```
Matcher m = p.matcher( str );
```

（3）使用匹配器对象 m 对文本字符串 str 进行词法分析或处理，例如查找或替换与正则表达式匹配的子串。

例 7-14 给出了一个在通讯录文本字符串中查找 E-mail 地址的 Java 演示程序。

例 7-14　在通讯录文本字符串中查找 E-mail 地址的 Java 演示程序（JMatcherFind.java）

```
1  import java.util.regex.*;              //导入 java.util.regex 包中与正则表达式相关的类
2  public class JMatcherFind {                //测试类
3      public static void main(String[] args) { //主方法
4          String str = "张三,A 公司,1391234567,zhangsan@cau.edu.cn\n" +
5                       "李四,B 公司,1397654321,li@163.com\n" +
6                       "王五,C 公司,1890000001,wuwang@sina.com\n";
                                              //通讯录字符串
7          String mf = "\\w+([-_]\\w+)*@\\w+([-_.]\\w+)*.\\w+";
                                              //E-mail 格式的正则表达式
8          Pattern p = Pattern.compile(mf);   //将正则表达式编译成模式对象
9          Matcher m = p.matcher(str);        //取得处理字符串 str 的匹配器对象
10         System.out.println( str +"\n 上述文本内容中的 E-mail 地址如下：" );
11         while (m.find() == true) {         //显示查找结果：检查是否还有下一个匹配项
12             int p1 = m.start();  int p2 = m.end();  //取出匹配项的起始和结束下标
13             System.out.println( p1 +"-" + p2 +": " + str.substring(p1, p2) );
                                              //显示匹配子串
14         }
15 } }
```

在 Eclipse 集成开发环境中运行例 7-14 的程序,其运行结果如图 7-16 所示。

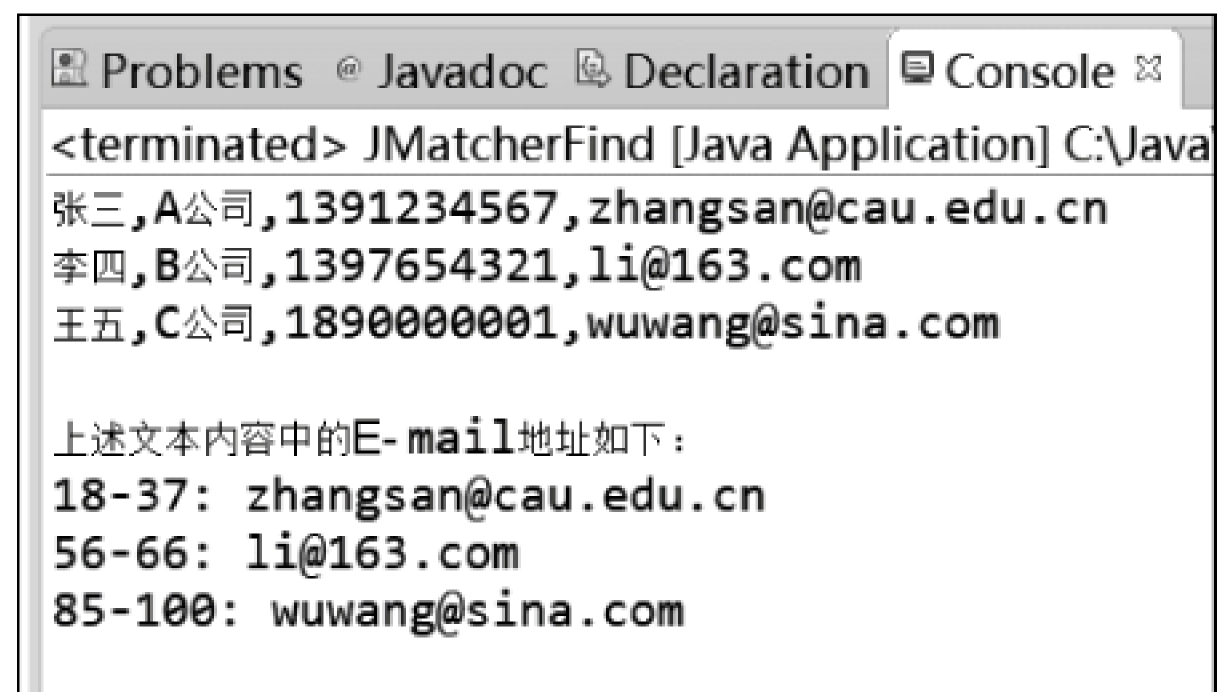


图 7-16 例 7-14 程序查找 E-mail 地址的结果

例 7-15 给出了一个使用匹配器类 Matcher 实现字符串查找与替换的 Java 演示程序。

例 7-15 一个使用匹配器类 Matcher 实现字符串查找与替换的 Java 演示程序(JMatcherReplace.java)

```
 1  import java.util.regex.*;                              //导入 java.util.regex 包中与正则
                                                           //表达式相关的类
 2  public class JMatcherReplace {                         //测试类
 3      public static void main(String[] args) {           //主方法
 4          //查找以下文本字符串 text 中的单词 dog(不区分大小写)
 5          String text = "I love small dog and big DOG.";
 6          Pattern p = Pattern.compile("dog", Pattern.CASE_INSENSITIVE); //对大小写不敏感
 7          Matcher m = p.matcher(text);                   //取得处理字符串 text 的匹配器对象
 8          //显示字符串中查找到的所有 dog(不区分大小写)
 9          System.out.println("搜索以下文本中的 dog(不区分大小写): " + text);
10          while (m.find() == true) {             //显示查找结果: 检查是否还有下一个匹配项
11              int s = m.start(); int e = m.end(); //取得匹配子串的起始和结束下标
12              String msg = String.format("%d-%d: %s", s, e, text.substring(s, e));
13              System.out.println(msg);
14          }
15          //将字符串中的 dog 全部替换成 cat
16          System.out.print("\n将 dog 替换成 cat: ");
17          String str = m.replaceAll( "cat" );            //将查找到的 dog 全部替换成 cat
18          System.out.println(str);
19  } }
```

在 Eclipse 集成开发环境中运行例 7-15 的程序,其运行结果如图 7-17 所示。

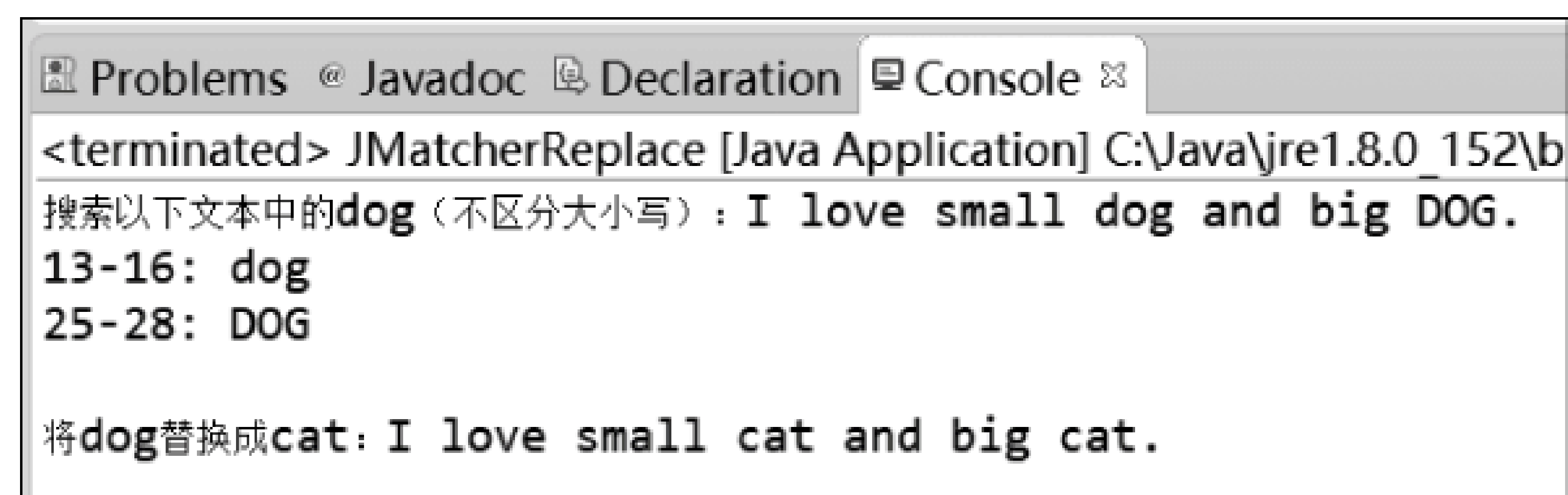


图 7-17 例 7-15 程序的字符串查找与替换结果

请读者阅读下面的匹配器类 Matcher 说明文档。

java. util. regex. **Matcher** 类说明文档

public final class **Matcher**
extends **Object**
implements **MatchResult**

	修　饰　符	类成员(节选)	功 能 说 明
1		boolean **find**()	查找下一个匹配项
2		int **start**()	取出匹配项的起始下标
3		int **end**()	取出匹配项的结束下标
4		boolean **matches**()	检查匹配器中的字符串是否符合正则表达式规定的模式
5		String **replaceAll**(String replacement)	替换所有的匹配项
6		Matcher **appendReplacement** (StringBuffer sb, String replacement)	取出匹配项及其前面的未匹配子串，替换匹配项，然后一起追加到 sb 中
7		StringBuffer **appendTail**(StringBuffer sb)	将匹配项后面的字符串追加到 sb 中
8		Matcher **usePattern**(Pattern newPattern)	重新指定正则表达式
9		boolean **lookingAt**()	匹配查找
10		int **groupCount**()	获得分组个数
11		String **group**(int group)	取出分组

…

本节习题

1. 字符串类 String 中的分词方法是(　　)。
   A. split()　　B. indexOf()　　C. substring()　　D. trim()
2. 字符串类 String 中取子字符串的方法是(　　)。
   A. split()　　B. indexOf()　　C. substring()　　D. trim()
3. 字符(　　)不符合正则表达式“[xyz]”所描述的模式。
   A. x　　B. y　　C. z　　D. 9
4. 字符(　　)不符合正则表达式“[x-z]”所描述的模式。
   A. x　　B. y　　C. z　　D. 9
5. 字符(　　)符合正则表达式“[^x-z]”所描述的模式。
   A. x　　B. y　　C. z　　D. 9

7.6　图像处理

使用 Java API 进行图像处理主要涉及以下几方面的内容。

(1) **打开图像文件**。图像有不同的文件格式，常用的有 JPEG、BMP、PNG、GIF 和 TIFF 等。打开图像文件就是将文件中的图像数据读入内存，以便后续显示或处理。Java

API 提供了两个存储图像数据的类:一个是不可修改的图标类 javax. swing. **ImageIcon**,主要用于显示;另一个是可以修改的带缓存图像类 java. awt. image. **BufferedImage**,主要用图像编辑或处理。

(2) **显示图像**。可以使用标签组件 JLabel 来显示图标类图像。显示带缓存的图像一般使用画布类 Canvas,通过重写绘图方法 paint()来显示图像。

(3) **修改图像**。可以修改带缓存的图像,例如修改像素值,或在图像上绘图。

(4) **保存图像文件**。通常需要将修改后的带缓存图像保存成图像文件。Java API 提供了一个图像输入输出类 javax. imageio. **ImageIO**。

7.6.1 图标类 ImageIcon

例 7-16 给出一个图标类 **ImageIcon** 的 Java 演示程序。该程序演示了一种最简单的打开并显示图像文件方法。

例 7-16 一个图标类 ImageIcon 的 Java 演示程序(JImageIconTest. java)

```
 1  import java.awt.*;                   //导入 java.awt 包中的类
 2  import java.awt.event.*;             //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                //导入 javax.swing 包中的类
 4
 5  public class JImageIconTest {                         //测试类
 6      public static void main(String[] args) {          //主方法
 7          JFrame w = new JFrame();                      //创建框架窗口类 JFrame 的对象
 8          w.setTitle("图像演示程序");                    //初始化窗口
 9          w.setSize(420, 320);  w.setLocation(100, 100);  w.setVisible(true);
10          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
11          //在窗口内容面板里添加一个显示图像的标签组件
12          Container cp = w.JgetContentPane();          //获得窗口的内容面板(默认边框布局)
13          JLabel box = new JLabel();
14          cp.add(box, BorderLayout.CENTER);             //将标签放在内容面板的中间
15          cp.validate();                                //检查并自动布局容器里的组件
16          //从图像文件加载图像,创建一个图标对象
17          ImageIcon ii = new ImageIcon("d:/1.jpg");
18          box.setIcon(ii);                              //在标签组件中显示图像
19  } }
```

在 Eclipse 集成开发环境中运行例 7-16 的程序,其运行结果如图 7-18 所示。



图 7-18 例 7-16 程序所显示出的图像

请读者阅读下面的图标类 ImageIcon 说明文档。

javax.swing.**ImageIcon** 类说明文档

public class **ImageIcon**
extends **Object**
implements **Icon**, **Serializable**, **Accessible**

	修 饰 符	类成员(节选)	功 能 说 明
1		**ImageIcon**()	构造方法
2		**ImageIcon**(String filename)	构造方法(从文件加载)
3		**ImageIcon**(URL location)	构造方法(从网络加载)
4		int **getIconWidth**()	返回图标的宽度
5		int **getIconHeight**()	返回图标的高度
6		Image **getImage**()	读出图标里的图像
7		void **setImage**(Image image)	设置图标的图像
…			

7.6.2 带缓存图像类 BufferedImage

例 7-17 给出一个带缓存图像类 **BufferedImage** 的 Java 演示程序。该程序首先打开与图 7-18 相同的图像文件，然后将图像修改成具有底片效果的补色图像，显示并将其保存成一个新的图像文件。

例 7-17 带缓存图像类 BufferedImage 的 Java 演示程序(JBufferedImageTest.java)

```
 1  import java.awt.*;                            //导入 java.awt 包中的类
 2  import java.awt.event.*;                      //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                         //导入 javax.swing 包中的类
 4  import java.io.*;                             //导入 java.io 包中的类
 5  import java.awt.image.BufferedImage;          //导入 java.awt.image 包中的类 BufferedImage
 6  import javax.imageio.ImageIO;                 //导入 javax.imageio 包中的类 ImageIO
 7
 8  public class JBufferedImageTest {             //测试类
 9      public static void main(String[] args) {  //主方法
10          JFrame w = new JFrame();              //创建框架窗口类 JFrame 的对象
11          w.setTitle("图像演示程序");             //初始化窗口
12          w.setSize(420, 320);   w.setLocation(100, 100);   w.setVisible(true);
13          w.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
14          //从文件加载图像,创建一个带缓存的图像对象
15          BufferedImage bi = null;
16          try {                                 //必须处理勾选异常 IOException
17              File fin = new File("d:/1.jpg");
18              bi = ImageIO.read(fin);           //图像输入输出类 ImageIO
19          }
20          catch(IOException e)                  //处理 IOException 异常(勾选异常)
21          {  System.out.println( e.getMessage() );   return;   }
22          //修改图像: 将图像中各像素的颜色值设为其补色
23          Color c1, c2;
```

```
24          for (int y = 0; y < bi.getHeight(); y++){        //行循环
25              for (int x = 0; x < bi.getWidth(); x++) {   //列循环
26                  c1 = new Color( bi.getRGB(x, y) );       //取出颜色值
27                  c2 = new Color( 255 - c1.getRed(), 255 - c1.getGreen(), 255 - c1.getBlue() );
28                  bi.setRGB(x, y, c2.getRGB());            //设为补色 c2
29          }}
30          //在窗口内容面板里添加一个显示图像的画布组件
31          Container cp = w.getContentPane();  //获得窗口的内容面板(默认边框布局)
32          MyCanvas cv = new MyCanvas(bi);     //创建用于显示图像的画布对象
33          cp.add(cv, BorderLayout.CENTER);    //将画布放在内容面板的中间
34          cp.validate();                      //检查并自动布局容器里的组件
35          //保存图像: 将修改后的图像保存到一个新的图像文件
36          try {                               //必须处理勾选异常 IOException
37              File fout = new File("d:/1-new.jpg");
38              ImageIO.write(bi, " jpg", fout); //图像输入输出类 ImageIO
39          }
40          catch(IOException e){  System.out.println( e.getMessage() );  }
41  }  }
42
43  class MyCanvas extends Canvas {             //定义一个新的画布类,重写 paint()方法
44      private  BufferedImage bi  = null;
45      public MyCanvas(BufferedImage i)        //构造方法
46      {  bi = i;  }
47      public void paint(Graphics g)           //重写 paint()方法,显示图像
48      {  if (bi != null)g.drawImage(bi, 0, 0, null);  }
49  }
```

在 Eclipse 集成开发环境中运行例 7-17 的程序,其运行结果如图 7-19 所示。



图 7-19　例 7-17 程序所显示出具有底片效果的反色图像

请读者阅读下面的带缓存图像类 BufferedImage 说明文档。

java.awt.image.**BufferedImage** 类说明文档

public class **BufferedImage**
extends **Image**
implements **WritableRenderedImage**, **Transparency**

	修饰符	类成员(节选)	功能说明
1	static	int **TYPE_INT_RGB**	图像常量,8 位 RGB 存储
2	static	int **TYPE_INT_BGR**	图像常量,8 位 BGR 存储

续表

	修　饰　符	类成员(节选)	功 能 说 明
3		**BufferedImage**(int width, int height, int imageType)	构造方法
4		int **getWidth**()	获取图像宽度
5		int **getHeight**()	获取图像高度
6		int **getType**()	获取图像类型
7		int **getRGB**(int x, int y)	读取某个像素的 RGB 值
8		void **setRGB**(int x, int y, int rgb)	设置某个像素的 RGB 值
9		BufferedImage **getSubimage**(int x, int y, int w, int h)	取出一幅子图像,即裁剪
10		Graphics **getGraphics**()	取出图像的绘图对象
		…	

图像输入输出类 **ImageIO** 定义了一组静态方法,其中最主要的两个方法是 **read**()和 **write**()。方法 read()可以从图像文件中加载数据,将其放入内存的图像对象。方法 write()则是将内存图像对象中的数据保存成一个图像文件。请读者阅读下面的图像输入输出类 ImageIO 说明文档。

javax. imageio. **ImageIO** 类说明文档

public final class **ImageIO**
extends **Object**

	修　饰　符	类成员(节选)	功 能 说 明
1	static	BufferedImage **read**(File input)	从文件读取图像
2	static	BufferedImage **read**(URL input)	从网址读取图像
3	static	BufferedImage **read**(InputStream input)	从输入流读取图像
4	static	boolean **write**(RenderedImage im, String formatName, File output)	将图像写入文件
6	static	boolean **write**(RenderedImage im, String formatName, OutputStream output)	将图像写入输出流
7	static	String[] **getReaderFormatNames**()	以字符串数组的形式返回系统可以读取的图像格式
8	static	String[] **getWriterFormatNames**()	以字符串数组的形式返回系统可以输出的图像格式
9	static	ImageWriter **getImageWriter**(ImageReader reader)	返回对应的编码器
10	static	ImageReader **getImageReader**(ImageWriter writer)	返回对应的解码器
		…	

7.6.3　修改图像

加载到内存的带缓存图像可以修改。图像修改一般分为两类:一类是**图像处理**,例如图像滤波、图像分割、γ 校正、几何变换等,编写图像处理程序需具备数字图像处理的专业知识;另一类是**图像编辑**,例如图像裁剪、为图像添加文字或在图像上绘图等。使用带缓存图像类 BufferedImage 的方法成员 **getSubimage**()取出指定区域的子图像,就可以实现图像裁剪的功能。使用图形类 **Graphics** 则可以实现在图像上添加文字或绘制图形的功能,每个带

缓存图像都包含一个 Graphics 类的子类绘图对象。

在带缓存图像上添加文字或绘制图形的方法是：首先取出带缓存图像的绘图对象，然后使用绘图对象在图像上绘图。例如，下面这段代码演示了在带缓存图像 bi 上添加文字"Hello, World!"，然后再画一个椭圆。

```
Graphics g = bi.getGraphics();                       //获取图像 bi 的绘图对象
g.setColor( Color.YELLOW );                          //设置绘图颜色
Font ef = new Font("TimesRoman", Font.PLAIN, 32);    //选择字体
g.setFont( ef );                                     //设置字体
g.drawString("Hello, World!", 20, 40);               //显示文字信息
g.drawOval(20, 80, 180, 60);                         //画一个椭圆
```

如果将这段代码插入到例 7-17 中代码第 29 行的后面，运行程序将显示如图 7-20 所示的绘图效果。



图 7-20　在图 7-19 的反色图像上添加文字和椭圆

本节习题

1. 下列 Java API 类中，与图像无关的类是(　　)。
   A. ImageIcon　　B. BufferedImage　　C. Image　　D. AudioClip
2. 下列 Java API 类中，(　　)能存储可被修改的图像数据。
   A. ImageIcon　　B. BufferedImage　　C. Image　　D. AudioClip
3. 带缓存图像类 BufferedImage 被定义在 Java API 包(　　)当中。
   A. java.awt　　B. java.awt.image
   C. javax.swing　　D. javax.imageio
4. 带缓存图像类 BufferedImage 中取子图像的方法是(　　)。
   A. getRGB()　　B. getType()
   C. getSubimage()　　D. getGraphics()
5. 图像输入输出类 ImageIO 中加载图像文件的方法是(　　)。
   A. read()　　B. write()
   C. getImageReader()　　D. getImageWriter()

7.7 声音处理

本节讲解如何编写简单的**录音**及**播放**程序。录音就是对麦克风输入的声音进行数字化采样，然后将其保存成音频文件。播放就是播放音频文件里的声音。

数字化采样所得到的声音数据可以做很多后续处理，例如语音识别。编写语音识别程序还需要语音信号处理、人工智能方面的专业知识，这超出了本书的范畴。请读者记住一点：数字化采样是后续声音信号处理的第一步。

7.7.1 相关概念与术语

1. 音频格式

音频格式涉及**编码算法**（例如 PCM 或 MP3）、**采样率**（例如 8kHz 或 16kHz）、**采样位数**（例如 8 位或 16 位）、**声道数**（例如单声道或双声道）、**帧率**、**每帧字节数**等参数。Java API 主要支持 **PCM** 音频编码算法，并提供了一个音频格式类 **AudioFormat** 来描述音频格式。创建音频格式对象时需指定相关的参数。例如：

```
AudioFormat af = new AudioFormat(8000f, 16, 1, true, true);  //音频格式：8kHz/16 位/单声道
```

请读者阅读下面的音频格式类 AudioFormat 说明文档。

javax. sound. sampled. **AudioFormat** 类说明文档

public class **AudioFormat**
extends **Object**

	修 饰 符	类成员（节选）	功 能 说 明
1		**AudioFormat**(float sampleRate, int sampleSizeInBits, int channels, boolean signed, boolean bigEndian)	构造方法
2		**AudioFormat**(AudioFormat. Encoding encoding, float sampleRate, int sampleSizeInBits, int channels, int frameSize, float frameRate, boolean bigEndian)	构造方法
3		int **getChannels**()	获取声道数
4		AudioFormat. Encoding **getEncoding**()	获取编码格式
5		float **getFrameRate**()	获取帧率
6		int **getFrameSize**()	获取每帧的字节数
7		float **getSampleRate**()	获取采样率
8		int **getSampleSizeInBits**()	获取采样位数

…

2. 音频文件格式

常用的音频文件格式有 WAVE、AIFF、AU、MP3、WMA 等，Java API 支持其中的 **WAVE**、**AIFF** 和 **AU** 文件格式。

3. 音频系统

Java API 需要基于本地计算机的音频系统(例如音频设备及其驱动程序)才能实现声音处理功能。为此,Java API 提供一个音频系统类 **AudioSystem**,其中定义了一组静态方法,用于访问本地计算机的音频系统。请读者阅读下面的音频系统类 AudioSystem 说明文档。

javax. sound. sampled. **AudioSystem** 类说明文档

public class **AudioSystem**
extends **Object**

	修 饰 符	类成员(节选)	功 能 说 明
1	static	boolean **isLineSupported**(Line. Info info)	系统是否支持某种数据线
2	static	boolean **isFileTypeSupported** (AudioFileFormat. Type fileType)	系统是否支持某种音频文件格式
3	static	boolean **isFileTypeSupported** (AudioFileFormat. Type fileType, AudioInputStream stream)	系统是否支持某种音频文件格式和某种音频输入流格式
4	static	TargetDataLine **getTargetDataLine** (AudioFormat format)	创建输入音频的目标数据线对象
5	static	SourceDataLine **getSourceDataLine** (AudioFormat format)	创建输出音频的源数据线对象
6	static	Clip **getClip**()	创建播放音频的片段对象
7	static	AudioInputStream **getAudioInputStream**(File file)	创建音频文件输入流对象
8	static	AudioInputStream **getAudioInputStream**(URL url)	创建网络音频文件输入流对象
9	static	int **write**(AudioInputStream stream, AudioFileFormat. Type fileType, File out)	保存音频文件
10	static	Mixer **getMixer**(Mixer. Info info)	创建混音器对象
11	static	Mixer. Info[] **getMixerInfo**()	获取混音器信息

…

4. 数据线

Java API 将连接音频设备的物理线路抽象成若干个**数据线**(data line)接口,如图 7-21 所示。图 7-21 中的目标数据线接口 **TargetDataLine** 表示连接音频输入设备(例如麦克风)的数据线,源数据线接口 **SourceDataLine** 接口表示连接音频输出设备(例如音箱)的数据线,片段接口 **Clip** 表示可以播放的音频数据(即音频片段)。

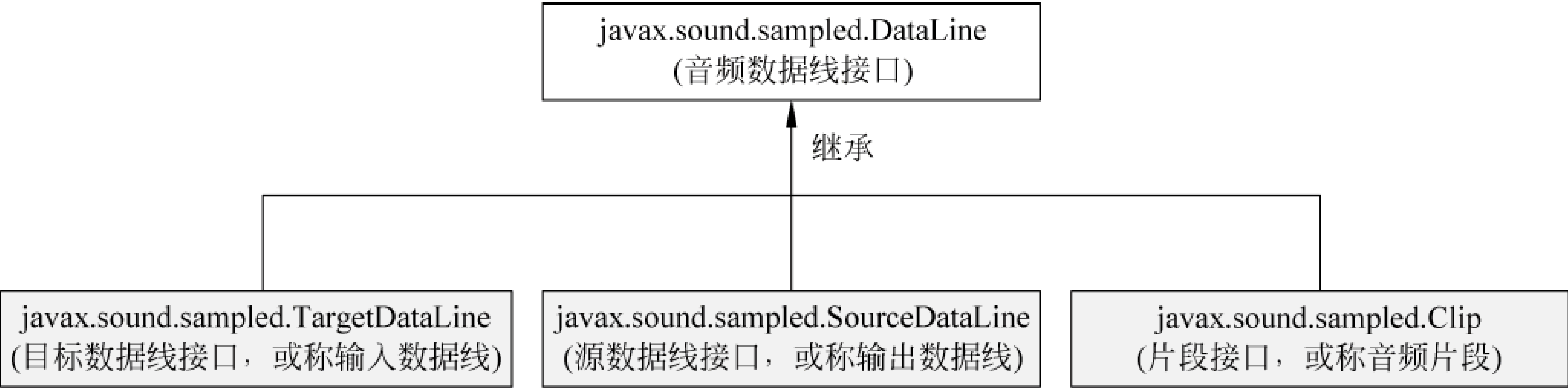


图 7-21　Java API 中定义的数据线接口

Java API 将与音频数字化采样相关的接口和类定义在 javax. **sound.** sampled 包中。

5. 音频输入流

打开或保存音频文件时需要用到 Java API 中的音频输入流类 **AudioInputStream**。这是一个包装类，它将音频数据包装成一个音频输入流对象。

7.7.2 录音

例 7-18 给出一个实现音频"录音-回放-保存"功能的 Java 演示程序。请读者仔细阅读其中的程序代码，这样才能了解录音的完整过程及其实现细节。

例 7-18 一个实现音频"录音-回放-保存"功能的 Java 演示程序(JRecordWavTest. java)

```
1   import java.io. * ;                              //导入 java.io 包中的输入输出流类
2   import javax.sound.sampled. * ;                  //导入 javax.sound.sampled 包中的音频类
3
4   public class JRecordWavTest {                    //主类
5       public static void main(String[ ] args) {    //主方法
6           AudioFormat af;                          //音频格式类 AudioFormat
7           TargetDataLine tLine;                    //目标(输入)数据线接口 TargetDataLine
8           SourceDataLine sLine;                    //源(输出)数据线接口 SourceDataLine
9           byte buf[ ] = new byte[1024 * 100];      //分配 100KB 数组用于保存录音数据
10          //先创建音频格式对象,指定音频格式
11          af = new AudioFormat(8000f, 16, 1, true, true);   //参数: 8kHz/16 位/单声道
12          //录音 - 回放试听 - 保存文件(.wav)
13          try {   //录音过程中可能会抛出勾选异常
14              //录音: 创建输入音频的目标数据线,然后打开录音,结束后停止并关闭
15              DataLine.Info info = new DataLine.Info(TargetDataLine.class, af);
16              if ( !AudioSystem.isLineSupported(info) )
17              {   System.out.println( "Line not supported" );   return;   }
18              tLine = ( TargetDataLine) AudioSystem.getLine(info);
                                                 //获取目标数据线
19              //开始录音: 打开并启动目标数据线,将音频数据存入字节数组 buf
20              System.out.println( "Line Start" );
21              tLine.open(af);                      //按照指定的音频格式 af 打开目标数据线
22              tLine.start();                       //启动目标数据线,开始录音
23              int dataLen = tLine.read(buf, 0, buf.length);
                                                 //读音频数据,读满缓冲区
24              tLine.stop();   tLine.close(); //停止录音,关闭目标数据线
25              System.out.println( "Line Stop" );
26              //回放: 创建输出音频的的源数据线,然后打开回放,结束后关闭
27              info = new DataLine.Info( SourceDataLine.class, af);
28              sLine = ( SourceDataLine) AudioSystem.getLine(info);   //获取源数据线
29              System.out.println( "Play" );
30              sLine.open(af);                      //按照指定的音频格式 af 打开源数据线
31              sLine.start();                       //启动源数据线,开始播放
32              sLine.write(buf, 0, dataLen);   //将音频数据写入源数据线
```

```
33              sLine.drain();  sLine.close();  //播放完音频数据后关闭源数据线
34              //保存：先将音频数据包装成一个字节数组输入流，
35              //然后再包装成一个音频输入流，最后保存到音频文件(.wav)
36              ByteArrayInputStream inBuf = new ByteArrayInputStream( buf);
37              AudioInputStream ais = new AudioInputStream(
38                                  inBuf, af,dataLen / af.getFrameSize() );
39              System.out.println( "Save" );
40              File fout = new File("d:/1.wav");//音频文件
41              AudioSystem.write(ais, AudioFileFormat.Type.WAVE, fout);  //输出音频文件
42              ais.close();  inBuf.close();    //关闭音频输入流和字节数组输入流
43          }
44          catch(LineUnavailableException e) {  System.out.println( e.getMessage() );  }
45          catch(IOException e) {  System.out.println( e.getMessage() );  }
46  }  }
```

7.7.3 播放音频文件

播放音频文件，首先需要为音频文件建立起音频输入流，然后在音频片段 Clip 对象中打开并播放输入流中的音频。例 7-19 给出一个播放音频文件的 Java 演示程序。

例 7-19 一个播放音频文件的 Java 演示程序(JPlayWavTest.java)

```
1   import java.io.*;                               //导入 java.io 包中的输入输出流类
2   import javax.sound.sampled.*;                   //导入 javax.sound.sampled 包中的音频类
3
4   public class JPlayWavTest {                     //主类
5       public static void main(String[] args) {    //主方法
6           try {  //播放过程中可能会抛出勾选异常
7               //为音频文件建立起音频输入流
8               File f = new File("d:/1.wav");          //音频文件
9               AudioInputStream ais = AudioSystem.getAudioInputStream(f);
10              System.out.println( ais.getFormat() ); //显示音频格式
11              Clip c = AudioSystem.getClip();         //获取播放音频的片段对象
12              c.open(ais);                            //在片段对象中打开音频输入流 ais
13              c.setFramePosition(0);                  //设置播放起始位置
14              System.out.println("Start");
15              c.start();                              //开始播放音频
16              Thread.sleep(10000);    //休眠等待 10 秒，否则音频未播放完程序就结束了
17              c.close();                              //关闭片段对象
18              System.out.println("Close");
19          }
20          catch(Exception e) {  System.out.println( e.getMessage() );  }
21  }  }
```

本节习题

1. 音频格式中没有包含参数(　　)。
   A. 编码算法　　B. 采样率　　C. 声道数　　D. 分辨率
2. 不属于音频文件的是(　　)。
   A. WAVE　　B. AIFF　　C. AU　　D. BMP
3. 表示连接音频输入设备(例如麦克风)的数据线接口是(　　)。
   A. TargetDataLine　　B. SourceDataLine　　C. Clip　　D. DataLine
4. 表示连接音频输出设备(例如音箱)的数据线接口是(　　)。
   A. TargetDataLine　　B. SourceDataLine　　C. Clip　　D. DataLine
5. 表示可以播放的音频片段接口是(　　)。
   A. TargetDataLine　　B. SourceDataLine
   C. Clip　　D. DataLine

本章学习要点

- Java API 中的类往往经历了多级抽象和多层包装,例如输入输出流类族中的类。读者在学习 Java API 过程中要注意及时总结并梳理出类与类之间的继承或包装关系。
- 初学者可以从常用类开始,先学习使用,然后再追溯其超类,逐步从微观到宏观,最终实现从整体上把握 Java API 类库的目标。
- 学习并掌握标准 I/O、文件 I/O 的常规编程方法和代码框架。
- 学习并掌握基本的文本处理方法,并能运用简单的正则表达式进行文本分析和处理。
- 学习并了解基本的图像及声音处理方法。

本章习题

1. 重写程序。阅读并重写 7.1.2 节例 7-1 和 7.1.3 节例 7-2 的键盘输入程序,然后键盘输入测试数据,通过比对显示结果来理解字节型和字符型输入流之间的区别。

2. 编写程序。使用标准 I/O 的格式化输入输出功能编写一个计算圆形、长方形面积和周长的 Java 程序。

3. 重写程序。阅读并理解 7.3.5 节例 7-7 中格式化输入输出文本文件的 Java 演示程序,然后重写这个程序。

4. 重写程序。阅读并理解例 7.4.3 节 7-9 中序列化及反序列化钟表对象的 Java 演示程序,然后重写这个程序。

5. 重写程序。阅读并理解 7.5.1 节例 7-10 中的文本编辑器演示程序,然后重写这个程序。

第8章 多线程并发编程

计算机可以同时运行多个程序，这样就能在同一台计算机上同时做不同的事情。例如，用户可以同时运行浏览器程序和多媒体播放器程序，这样就能一边浏览网页，一边听音乐。

在单个CPU上同时运行多个程序将采用**分时**(time-sharing)技术。把CPU的运行时间划分成很小的**时间片**(time slice)，然后按时间片轮流执行各程序。如果程序在用完一个时间片之后未能完成执行，则被暂时挂起，等待下一轮继续执行。

因为计算机执行速度很快，每个程序被挂起的时间很短，用户在感觉上似乎是多个程序在同时运行。采用分时技术同时执行多个程序的方式称为**并发**(concurrency)。操作系统全权负责管理和调度多个程序的并发执行，程序员在编程时不需要做什么事情。

本章学习**多线程并发编程**，其内容是如何让单个程序同时做多件事情，即在同一进程内再创建多个**线程**，然后并发执行它们。例如，如何让一个音乐播放程序能够在下载网络音乐的同时播放它，边下载边播放而不是一定要等下载完之后才播放。线程需要程序员编写代码来创建和执行。

8.1 多线程并发程序

本节讲解进程与线程、单线程与多线程等基本概念，然后通过具体的程序实例让读者了解什么是多线程并发程序。

8.1.1 进程与线程

1. 进程

操作系统为每个加载到内存执行的程序创建一个**进程**(process)。进程可理解为是一个**运行环境**(execution environment)，具有运行程序所需的计算资源和存储资源。每个进程运行一个程序，多个进程就可以同时运行多个程序，这就是**进程并发**。多个进程通过**分时**技术分享CPU的计算资源，通过**地址空间映射**技术分享内存的存储资源。

操作系统全权负责进程的创建、管理、调度和删除，并为进程分配CPU时间片和内存空间。程序员可以忽略进程的存在，编程时通常不需要为进程做什么事情。

2. 线程

程序可能需要完成比较复杂的任务，可以将复杂任务分解成多个小的子任务。假设一

个音乐播放程序需要完成下载并播放音乐的任务，可以将这个任务分解成下载和播放两个独立的子任务。

如果将完成子任务的指令序列（即语句序列）称作一个**算法**，那么一个程序可以包含多个算法。通常，程序中的多个算法是按顺序依次执行的，称为**串行**执行。例如，音乐播放程序包含“下载”和“播放”两个算法，执行时应当先执行“下载”算法，下载完成后再执行“播放”算法。

程序可以为其中的算法单独创建**线程**（thread），每个线程负责执行一个算法。多个线程之间各自独立运行，通过分时技术可以同时执行多个算法，这就是**线程并发**。一个多线程并发程序在运行时看起来就像是同时在做多件事情。例如，音乐播放程序可以为下载算法和播放算法分别创建线程，然后通过分时技术同时执行这两个算法，这样就能边下载，边播放。

3. 进程与线程的关系

进程包含线程。程序所创建的线程包含于运行该程序的进程之中。一个线程可理解为是包含于进程内部的一个可独立运行算法的运行环境。在进程中创建线程，其目的是对进程的**计算资源**做进一步细分。程序员可运用线程技术对程序做更直接、更精细的并发控制。

一个进程可以包含多个线程。同一线程的算法内部是**串行**执行的，而不同线程的算法之间则是通过分时技术**并发**执行的。每个线程所执行的算法是一个指令序列，将其称作一个**指令执行流**。一个线程包含一个指令执行流。一个进程中可以有多个线程，因此一个进程可能包含多个并发执行的指令执行流。

进程还包含**存储资源**，这些存储资源可以被进程中各线程所运行的算法**共享**。不同线程的算法之间虽然是各自独立执行的，但它们需要协同工作。通过访问共同的存储资源，同一进程中不同线程的算法之间可以共享数据，进而实现多线程协同工作。

这里对进程和线程做如下总结。

- **一个进程运行一个程序**。操作系统为每个加载到内存执行的程序创建一个进程。
- **一个线程运行一个算法**。一个程序可以划分成多个算法，将算法分散交由不同的线程去执行，这样可以实现多线程并发执行。程序员负责创建线程，并为线程指定所要运行的算法。
- **一个进程可以包含多个线程**。同一进程中的多个线程之间虽然各自独立运行算法，但算法之间需要共享数据，这样才能实现多线程协同工作。

8.1.2　单线程串行程序

每个 Java 程序在运行时都会单独创建一个进程。该进程自动包含一个由系统创建的**主线程**（main thread），用于运行程序的主方法 main()。主线程从主方法的第一条语句开始执行，直到最后一条语句执行结束，或遇到 return 语句中途退出时为止。

同一线程的算法内部是串行执行的。如果程序员没有为程序额外创建线程，那么该程序在运行时将只有一个主线程。主线程按串行方式执行程序中的指令序列，这是一种**单线程串行**程序。本章之前各章节所编写的程序例子都属于单线程串行程序。

如果程序的主方法调用某个子方法，则主线程在执行到方法调用语句时将暂停主方法

的执行,转去执行子方法,执行结束后再返回主方法继续执行。这种程序仍然属于单线程串行程序,因为主方法与子方法的算法是在同一线程中按串行方式执行的。

例 8-1 给出一个模拟音乐播放器的单线程串行 Java 演示程序。

例 8-1 一个模拟音乐播放器的单线程串行 Java 演示程序(JPlayerST.java)

```
1  public class JPlayerST {                          //主类:单线程串行程序
2      public static  void main(String[] args) {  //主方法
3          //下载算法:模拟网络下载,显示 5 次信息"Downloading ..."
4          int count = 1;                            //显示计数
5          while (count <= 5) {                      //循环显示 5 次信息
6              System.out.println("Downloading ..." + count);
7              count++;
8              //每显示一次信息后让算法休眠(暂停)0.1 秒,模拟下载过程
9              try {                                 //捕获并处理可能抛出的勾选异常
10                 Thread.sleep(100);                //可能抛出勾选异常 InterruptedException
11             }
12             catch (InterruptedException e) {      //捕捉并处理异常
13                 System.out.println( e.getMessage() );
14                 return;                           //退出主方法,程序结束
15             }
16         }
17         play();                                   //下载完成后,调用子方法 play(),模拟播放音乐
18     }
19
20     static  void play() {                         //子方法
21         //播放算法:模拟播放音乐,显示 5 次信息"Playing ..."
22         int count = 1;                            //显示计数
23         while (count <= 5) {                      //循环显示 5 次信息
24             System.out.println("\t Playing ..." + count);
25             count++;
26             //每显示一次信息后让算法休眠(暂停)0.1 秒,模拟播放过程
27             try {                                 //捕获并处理可能抛出的勾选异常
28                 Thread.sleep(100);                //可能抛出勾选异常 InterruptedException
29             }
30             catch (InterruptedExceptione) {       //捕捉并处理异常
31                 System.out.println( e.getMessage() );
32                 return;                           //退出子方法,返回主方法
33             }
34         }
35 } }
```

在 Eclipse 集成开发环境中运行例 8-1 的程序,运行结果如图 8-1 所示。

这里对例 8-1 的程序做如下两点说明。

(1) 单线程串行程序的执行流程。

例 8-1 程序中包含下载和播放两个算法,但它只有一个主线程。主线程以串行方式首先执行模拟网络下载的算法,显示 5 次“Downloading...”;然后再去执行模拟播放音乐的算法,显示 5 次“Playing...”。单线程串行程序在执行时只有一个指令执行流,图 8-2 给出了例 8-1 程序的执行流程示意图。可以看出,使用单线程串行程序播放音乐需要先等待,等音乐下载完成后才能播放。

```
Problems  Javadoc  Declaration  Console
<terminated> JPlayerST [Java Application] C:\Java\jre
Downloading ...1
Downloading ...2
Downloading ...3
Downloading ...4
Downloading ...5
         Playing ...1
         Playing ...2
         Playing ...3
         Playing ...4
         Playing ...5
```

图 8-1　例 8-1 程序的运行结果

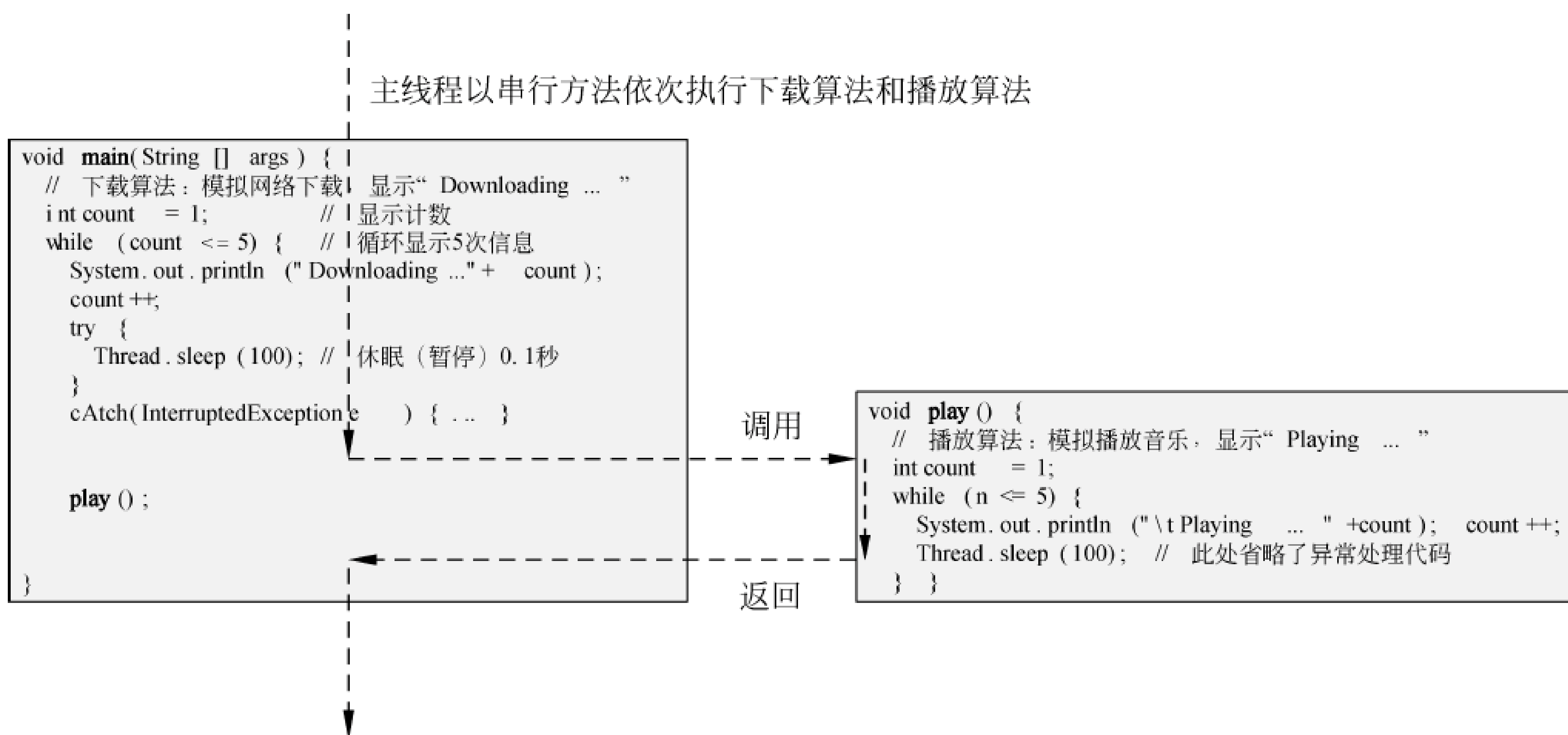


图 8-2　例 8-1 程序的执行流程示意图

(2) 线程的休眠。

为了模拟音乐下载的过程，例 8-1 程序在每次显示完信息“Downloading...”之后让下载算法休眠 0.1 秒，参见例 8-1 中代码第 9～15 行。所谓让下载算法休眠 0.1 秒，其含义是让执行该算法的主线程暂停 0.1 秒，然后再继续执行算法。休眠所使用的线程类 **Thread** 及其静态方法 **sleep**()将在后面讲解。这里需要说明的是，方法 sleep()在执行时可能会抛出勾选异常 **InterruptedException**。调用该方法必须对该异常进行捕获处理，否则程序编译不能通过。

同理，例 8-1 程序在模拟音乐播放的算法中也使用了休眠方法，参见例 8-1 中代码第 27～33 行。

8.1.3　多线程并发程序

例 8-1 的音乐播放器是一个单线程串行程序，其中的下载算法和播放算法都被安排在主线程中执行。同一线程中的算法只能串行执行，因此使用这个单线程串行程序播放音乐需要先等待，等音乐下载完成后才能播放。

如果希望改进例 8-1 的音乐播放器程序,让它能边下载,边播放,程序员就需要运用多线程并发编程技术,将下载算法和播放算法分别放入不同线程中去执行。

Java API 为多线程并发编程提供了相关的接口和类。其中最基本的有两个:一个是可运行接口 **Runnable**;另一个是线程类 **Thread**。程序员可以按如下步骤将算法单独放入一个线程,这样它就能与其他线程里的算法并发执行。

(1) **将算法封装成一个可运行的算法对象**。通过实现 Java API 的可运行接口 Runnable,程序员可以将算法封装成一个可被线程运行的算法对象。

(2) **创建线程对象,并在线程对象中运行算法对象**。每个 Java 程序在运行时都会由系统自动创建一个主线程,并在其中执行程序的主方法 main()。程序员可以使用 Java API 的线程类 Thread 创建新的线程对象,并在其中运行封装好的算法对象。由程序员创建的线程对象统称为**子线程**(sub thread)。子线程在启动后将与主线程并发执行,分头执行各自的算法。

一个程序可以包含多个线程,其中一个是系统自动创建的主线程,其余的则是由程序员创建的子线程,多个线程之间通过分时技术并发执行,这样的程序称为**多线程并发程序**。例 8-2 给出一个模拟音乐播放器的多线程并发 Java 演示程序。

例 8-2 一个模拟音乐播放器的多线程并发 Java 演示程序(JPlayerMT.java)

```
1   public class JPlayerMT {                                  //主类:多线程并发程序
2       public static  void main(String[] args) {             //主方法
3           //将播放算法放入单独的线程中去执行
4           PlayAlgorithm a = new PlayAlgorithm(); //创建一个可运行的播放算法对象 a
5           Thread t = new Thread(a);                         //新建子线程 t,在 t 中运行算法对象 a
6           t.start();                                        //启动子线程 t
7           //下面的下载算法在主线程中执行,将与上面子线程中的播放算法并发执行
8           //下载算法:模拟网络下载,显示 5 次信息"Downloading ..."
9           int count = 1;                                    //显示计数
10          while (count <= 5) {                              //循环显示 5 次信息
11              System.out.println("Downloading ..." + count);  count++;
12              //每显示一次信息后让程序休眠(暂停)0.1 秒,模拟下载过程
13              try {                                         //捕获并处理可能抛出的勾选异常
14                  Thread.sleep(100);                        //可能抛出勾选异常 InterruptedException
15              }
16              catch (InterruptedException e) {              //捕捉并处理异常
17                  System.out.println( e.getMessage() );
18                  return;                                   //退出主方法,程序结束
19              }
20          }
21          //当主线程和子线程都执行结束时则退出程序,进程也随之结束
22  } }
23
24  class PlayAlgorithm implements Runnable {                 //可运行的算法类 PlayAlgorithm
25      public void run() {                                   //描述算法的方法 run
26          //播放算法:模拟播放音乐,显示 5 次信息"Playing ..."
27          int count = 1;                                    //显示计数
28          while (count <= 5) {                              //循环显示 5 次信息
29              System.out.println("\t Playing ..." + count);  count++;
```

```
30              //每显示一次信息后让程序休眠(暂停)0.1 秒。模拟播放过程
31              try {                          //捕获并处理可能抛出的勾选异常
32                  Thread.sleep(100);         //可能抛出勾选异常 InterruptedException
33              }
34              catch (InterruptedException e) {   //捕捉并处理异常
35                  System.out.println( e.getMessage() );
36                  return;                    //退出子方法,返回主方法
37              }
38          }
39          //执行完算法代码后退出 run()方法,执行该方法的线程也随之结束
40  }  }
```

在 Eclipse 集成开发环境中运行例 8-2 的程序,运行结果如图 8-3 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> JPlayerMT [Java Application] C:\Java\jre
Downloading ...1
        Playing ...1
Downloading ...2
        Playing ...2
        Playing ...3
Downloading ...3
        Playing ...4
Downloading ...4
Downloading ...5
        Playing ...5
```

图 8-3 例 8-2 程序的运行结果

这里对例 8-2 的程序做如下两点说明。

(1) 多线程并发程序的执行流程。

例 8-2 的音乐播放器程序包含两个线程:一个是主线程,用于运行下载算法;另一个是子线程,它是在主线程中创建的,用于运行播放算法。多线程并发程序在执行时包含多个指令执行流,图 8-4 给出了例 8-2 程序的执行流程示意图,其中包含两个指令执行流,一个是主线程中的下载算法执行流,另一个是子线程中的播放算法执行流,它们是交替执行的。

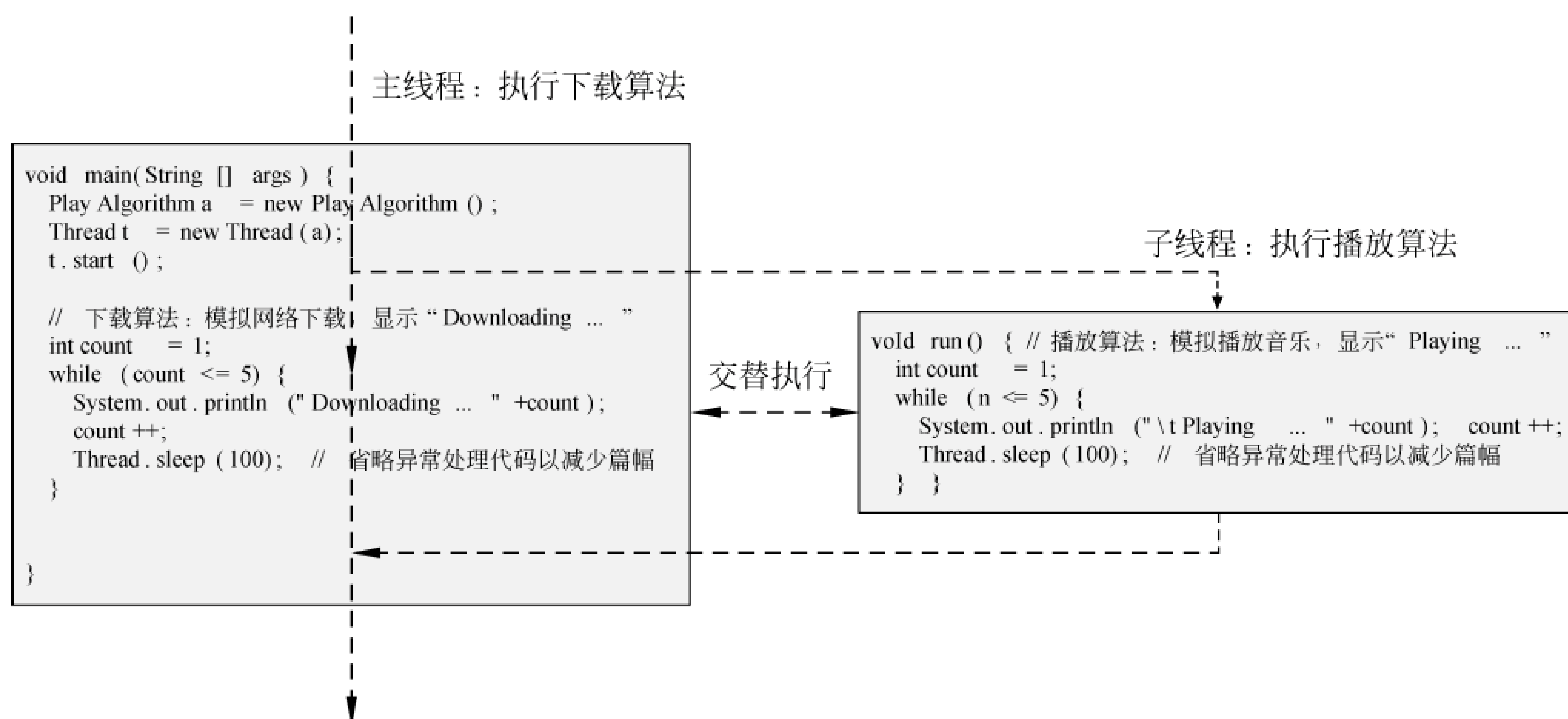


图 8-4 例 8-2 程序的执行流程示意图

(2) 多线程并发执行具有随机性。

并发执行多个线程,每个线程在什么时候执行、是先执行还是后执行,这就是线程的执行调度。Java 虚拟机负责线程的管理和执行调度。线程的执行调度具有随机性。例如,再次运行例 8-2 程序,所看到的运行结果可能与图 8-3 有所不同。

本节习题

1. 下列关于进程的描述中,错误的是(　　)。
   A. 操作系统为每个加载到内存执行的程序一次性创建多个进程
   B. 进程具有运行程序所需的计算资源和存储资源
   C. 多个进程通过分时技术分享 CPU 的计算资源
   D. 多个进程通过地址空间映射技术分享内存的存储资源
2. 下列关于线程的描述中,错误的是(　　)。
   A. 一个进程可以包含多个线程
   B. 同一线程的算法内部是串行执行的
   C. 不同线程的算法之间是并发执行的
   D. 同一进程中不同线程的算法之间不能共享数据
3. 一个进程至少包含(　　)个线程。
   A. 0　　B. 1　　C. 2　　D. 3
4. Java API 为多线程并发编程提供了一个接口 Runnable,该接口的作用是(　　)。
   A. 将算法封装成一个可被线程运行的算法对象
   B. 将算法封装成一个可独立运行的进程对象
   C. 创建线程并在线程中运行算法对象
   D. 创建进程并在进程中运行算法对象
5. Java API 为多线程并发编程提供了一个类 Thread,该类的作用是(　　)。
   A. 将算法封装成一个可被线程运行的算法对象
   B. 将算法封装成一个可独立运行的进程对象
   C. 创建线程并在线程中运行算法对象
   D. 创建进程并在进程中运行算法对象

8.2 多线程编程及并发调度

多线程并发编程过程中有 3 个非常重要的概念,它们分别是**算法**、运行算法的**线程**,以及多线程在执行时的**并发调度**。

8.2.1 算法

算法是程序中完成某种功能的指令序列(即语句序列),一个程序可以包含多个算法。如果将算法代码封装成可以被线程运行的算法对象,然后将它单独放入一个线程,那么这个算法就能与其他线程里的算法并发执行。

Java API 提供了一个"**可运行的**"接口 **Runnable**,用于将算法封装成可被线程运行的算法对象。请读者阅读下面的接口 Runnable 说明文档。

java. lang. **Runnable** 接口说明文档			
@FunctionalInterface public interface **Runnable**			
	修 饰 符	功能接口成员	功 能 说 明
1		void **run**()	描述需在线程中执行的算法代码

接口 Runnable 是一个功能接口，其中只包含一个抽象方法 **run**()，它为在线程中运行算法提供了一种统一的算法接口标准。

定义一个实现接口 Runnable 的算法类，在类中实现抽象方法 run()，编写需被并发执行的算法代码。用这个算法类所创建的对象就是可以被线程运行的算法对象。实现接口 Runnable 算法类的代码框架如下：

```
class 算法类名 implements Runnable {
    public  void run() {
        …                                   //此处编写需被并发执行的算法代码
    }
}
```

例 8-2 中代码第 24～40 行就是按上述代码框架编写的一个播放算法类 PlayAlgorithm。用这个类所创建的对象就是可以被线程运行的播放算法对象。例如：

```
PlayAlgorithm a = new PlayAlgorithm();  //创建一个可运行的播放算法对象 a
```

可以使用匿名类来创建算法对象，这样能简化程序代码。例如，改用匿名类的形式来直接创建上面的播放算法对象 a，其代码框架如下。

```
Runnable a = new Runnable() {           //改用匿名类直接创建一个播放算法对象 a
    public void run() {
        …                               //此处编写需并发执行的播放算法
    }
};
```

8.2.2 线程

算法需要放入线程才能并发执行。Java API 提供了一个线程类 **Thread**，用于创建线程对象。将算法对象放入线程对象，然后启动就可以并发执行算法了。请读者阅读下面的线程类 Thread 说明文档。

java. lang. **Thread** 类说明文档			
public class **Thread** extends **Object** implements **Runnable**			
	修 饰 符	类成员(节选)	功 能 说 明
1	static	int **MAX_PRIORITY**	优先级常量，最大优先级
2	static	int **MIN_PRIORITY**	优先级常量，最小优先级

续表

	修 饰 符	类成员(节选)	功 能 说 明
3	static	int **NORM_PRIORITY**	优先级常量,一般优先级
4	static	void **sleep**(long millis)	休眠,释放执行权
5	static	void **yield**()	释放执行权
6	static	Thread **currentThread**()	返回当前正在执行的线程
7	static	boolean **interrupted**()	检查当前线程是否有中止请求
8		**Thread**(Runnable target)	构造方法
9		**Thread**(Runnable target, String name)	构造方法
10		void **start**()	启动线程
11		void **run**()	描述在线程中执行的算法
12		void **setPriority**(int newPriority)	设置线程优先级
13		int **getPriority**()	获取线程优先级
14		void **setName**(String name)	设置线程名
15		String **getName**()	获取线程名
16		long **getId**()	获取线程号
17		boolean **isAlive**()	检查线程是否还未结束
18		void **interrupt**()	请求中止线程
19		Thread. State **getState**()	获取线程状态
20		void **setDaemon**(boolean on)	设置后台(守护)线程
21		boolean **isDaemon**()	检查是否是后台线程
		…	

一个多线程并发程序包含多个线程,其中一个是系统自动创建的主线程,其余的则是由程序员创建的子线程。每个线程都有**线程号**(ID)、**线程名称**(name)、**优先级**(priority)和**状态**(state)等属性。

调用线程对象的 **start**()方法启动线程,计算机将在线程中执行算法对象的 **run**()方法。该线程与所在进程的其他线程(包括主线程)一起并发执行。当执行完算法对象 run()方法中的所有代码,或遇到 return 语句中途退出时,线程即宣告结束。线程中的算法对象可以在执行过程中响应外部中断请求,并使用 return 语句退出 run()方法,结束线程。

当程序进程中的主线程和所有子线程都执行结束后,则退出程序,进程也随之结束。

可以将子线程设置为**守护**(daemon)线程,或称后台线程。只有当主线程及所有非守护子线程都结束后,进程才会结束。进程结束时,守护线程会自动结束。

1. 两种中途停止子线程的方法

例 8-3 给出一个线程类 Thread 的 Java 演示程序,其中包含一个主线程和两个子线程。这个例子演示了两种中途停止子线程的方法:一是通过线程类 Thread 的方法成员 **interrupt**()从外部中止运行;二是将子线程设为守护线程,守护线程在进程结束时将自动停止。

例 8-3 一个线程类 Thread 的 Java 演示程序(JThreadTest.java)

```
 1  public class JThreadTest {                               //主类:一个主线程 + 两个子线程
 2      public static  void main(String[] args) {            //主方法
 3          Thread t1 = new Thread( new Sub() );              //子线程 t1 运行算法 Sub
 4          Thread t2 = new Thread( new SubDaemon() );//子线程 t2 运行算法 SubDaemon
 5          t1.start();                                       //启动子线程 t1
 6          t2.setDaemon(true);  t2.start();                  //将 t2 设为守护线程,启动子线程 t2
 7          //下面的算法将与上面子线程中的算法并发执行
 8          int count = 1;                                    //显示计数
 9          while (count <= 5) {                              //循环显示 5 次信息
10              System.out.println("Main ..." + count);  count++;
11          }
12          if ( t1.isAlive() )                               //如果子线程 t1 还未运行结束,则请求中止
13              t1.interrupt();                               //请求中止子线程 t
14          //子线程 t2 是守护线程,在进程结束时将自动结束
15  }}
16
17  class Sub  implements Runnable {                          //算法类 1
18      public  void run() {                                  //描述算法的方法 run()
19          int count = 1;                                    //显示计数
20          while (true) {                                    //循环显示信息,死循环
21              System.out.println("\t Sub ..." + count);  count++;
22              if ( Thread.currentThread().interrupted() )  //检查当前线程是否有中止申请
23                  return;                                   //退出子线程
24          }
25  }  }
26
27  class SubDaemon  implements Runnable {                    //算法类 2
28      public  void run() {                                  //描述算法的方法 run()
29          int count = 1;                                    //显示计数
30          while (true) {                                    //循环显示信息,死循环
31              System.out.println("\t SubDaemon ..." + count);  count++;
32              try {                                         //捕获并处理可能抛出的勾选异常
33                  Thread.sleep(30);                         //休眠 0.03 秒
34              }
35              catch (InterruptedException e) {              //捕捉并处理异常
36                  System.out.println( e.getMessage() );  return;
37              }
38          }
39  }  }
```

在 Eclipse 集成开发环境中运行例 8-3 的程序,运行结果如图 8-5 所示。

2. 定义算法类时继承线程类 Thread

之前的例子在定义算法类时都是实现 Runnable 接口,然后定义其抽象方法 run(),编写需并发执行的算法代码。

线程类 Thread 也实现了 Runnable 接口,可以直接继承线程类 Thread 来定义算法类。例 8-4 给出一个通过继承线程类 Thread 来定义算法类的 Java 演示程序。

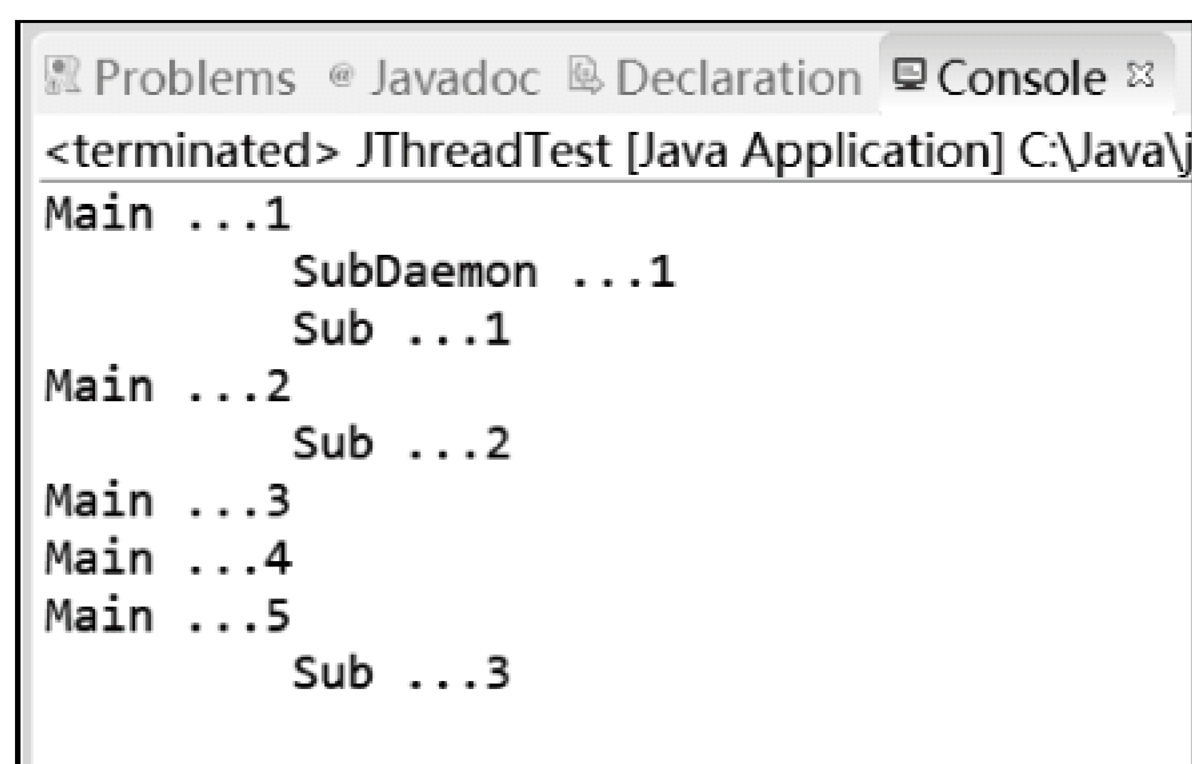


图 8-5　例 8-3 程序的运行结果

例 8-4　一个通过继承线程类 Thread 来定义算法类的 Java 演示程序(JSubThreadTest.java)

```
1  public class JSubThreadTest {                    //主类
2      public static void main(String[ ] args) {    //主方法
3          SubThread t = new SubThread();           //创建包含算法的线程对象 t
4          t.start();                               //启动线程,运行其中的算法
5      }
6  }
7
8  class SubThread  extends Thread {    //通过继承线程类 Thread 来定义算法类
9      public  void run() {             //重写 run()方法,编写需在线程中运行的算法代码
10         System.out.println("Hello from a thread!");
11     }
12 }
```

通过继承线程类 Thread 来定义算法类,实际上是将"算法"和"运行算法的线程"合二为一。需要注意的是,使用这种方法定义算法类时不能再继承其他类,因为 Java 语言只支持单继承。实际应用中,定义算法类更多的是实现接口 Runnable,而不是继承线程类 Thread。

8.2.3　多线程的并发调度

多线程并发程序在执行时,只有拿到 CPU 控制权的线程才能执行。Java 虚拟机负责线程的管理和执行调度。程序员无法掌控线程在什么时候执行,也很难精确控制哪个线程先执行,哪个线程后执行。

1. 线程的 5 种状态

新建的线程对象处于**新建**(new)状态。可以调用线程对象的 start()方法启动线程。线程启动后不是立即执行,而是进入**可运行**(runnable)状态,或称为**就绪**(ready)状态。进入可运行状态的线程需在 Java 虚拟机的**可运行队列**中排队,等待 CPU 控制权。

拿到 CPU 控制权的线程进入**运行**(running)状态,处于运行状态的线程被称为**当前线程**(current thread)。当前线程的运行时间只有一个 CPU 时间片,如果在这个时间片内执行完了算法对象 run()方法中的所有代码或遇到 return 语句中途退出时,线程即宣告结束,

进入结束状态；否则，交出 CPU 控制权，重新排队，等待下一次运行。

内存中的线程对象从创建到结束这个时间段称为线程对象的**生命周期**(life cycle)。一个线程对象可能处于 5 种不同的状态，它们分别是**新建**(new)状态、可运行状态、运行状态、**阻塞**(blocked)状态或**结束**(terminated)状态，如图 8-6 所示。一个线程对象在同一时刻只会处于这 5 种状态中的某一种状态。

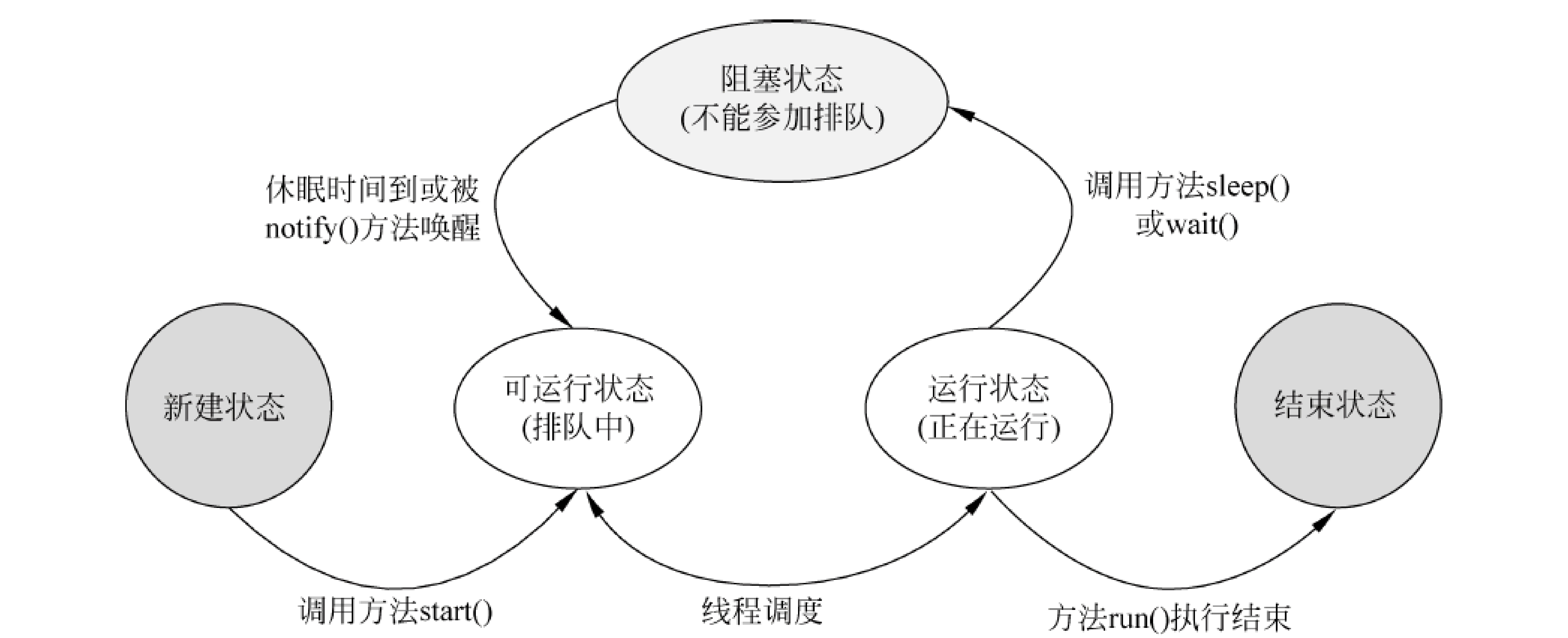


图 8-6　线程对象的 5 种不同状态

Java API 在线程类 Thread 中定义了一个内部类 **State**。这是一个枚举类型，其中定义了描述线程状态的枚举常量。

java. lang. **Thread. State** 内部类说明文档

public static enum **Thread. State**
extends **Enum** < **Thread. State** >

	修　饰　符	枚举常量(节选)	功 能 说 明
1		**NEW**	已创建但还未启动的线程
2		**RUNNABLE**	处于可运行(就绪)状态的线程
3		**BLOCKED**	处于阻塞状态的线程
4		**WAITING**	处于等待状态的线程
5		**TIMED_WAITING**	处于定时等待状态的线程
6		**TERMINATED**	已执行结束的线程
		…	

2. 线程对象的优先级

Java 虚拟机根据线程的优先级来决定哪个线程先执行，哪个线程后执行。线程优先级为 1～10，共分为 10 个等级。1 级为最低优先级，10 级为最高优先级，线程默认的优先级是 5 级。排队时，优先级高的线程对象优先获得 CPU 控制权，优先被运行。

在线程中创建另一个新线程时，新线程具有与当前线程相同的优先级。线程对象可以通过方法成员 setPriority()修改优先级，或通过 getPriority()获得自己当前的优先级。

3. 多线程与 CPU

在单核 CPU 上运行多线程程序时,操作系统会采用分时技术,并发执行各个线程。而在多核或多 CPU 的计算机上运行多线程程序时,操作系统会将线程分配到不同的运行核或 CPU 上运行,这样就能实现真正的**并行**(parallel)执行。

完成同样的功能,程序员在编程时可以采用单线程,也可以采用多线程。与单线程相比,采样多线程的程序在多核或多 CPU 计算机上的运行速度会成倍提高。

本节习题

1. “可运行的”接口 Runnable 中定义的方法是(　　)。
   A. run()　　B. start()
   C. sleep()　　D. setPriority()
2. 线程类 Thread 中没有定义的方法是(　　)。
   A. run()　　B. start()
   C. sleep()　　D. exit()
3. 线程属性中没有包含(　　)。
   A. 线程号　　B. 线程名称
   C. 优先级　　D. CPU 运行核的数量
4. 线程在启动后进入的状态是(　　)。
   A. 新建状态　　B. 可运行状态
   C. 运行状态　　D. 阻塞状态
5. 线程类 Thread 中将线程设为后台线程的方法是(　　)。
   A. setDaemon()　　B. setBackground()
   C. sleep()　　D. yield()

8.3 多线程之间的并发与互斥

一个程序可以划分成多个算法,将算法分散交由不同的线程去执行,这样就可以实现多线程并发执行。多线程并发程序的各个算法之间虽然都是独立执行的,但它们可能需要共享数据,这样才能实现多线程协同工作。

本节通过一个具体的程序实例来讲解为什么需要使用多线程并发、并发时为什么要共享数据,以及多线程共享数据时需要考虑的问题。

8.3.1 单线程售票服务演示程序

假设编写一个销售火车票或飞机票的计算机售票服务程序,图 8-7 描述了该程序的应用场景。这个售票场景比较简单,只有一个售票窗口,客户需排队购票。售票窗口的服务流程可分为如下两步。

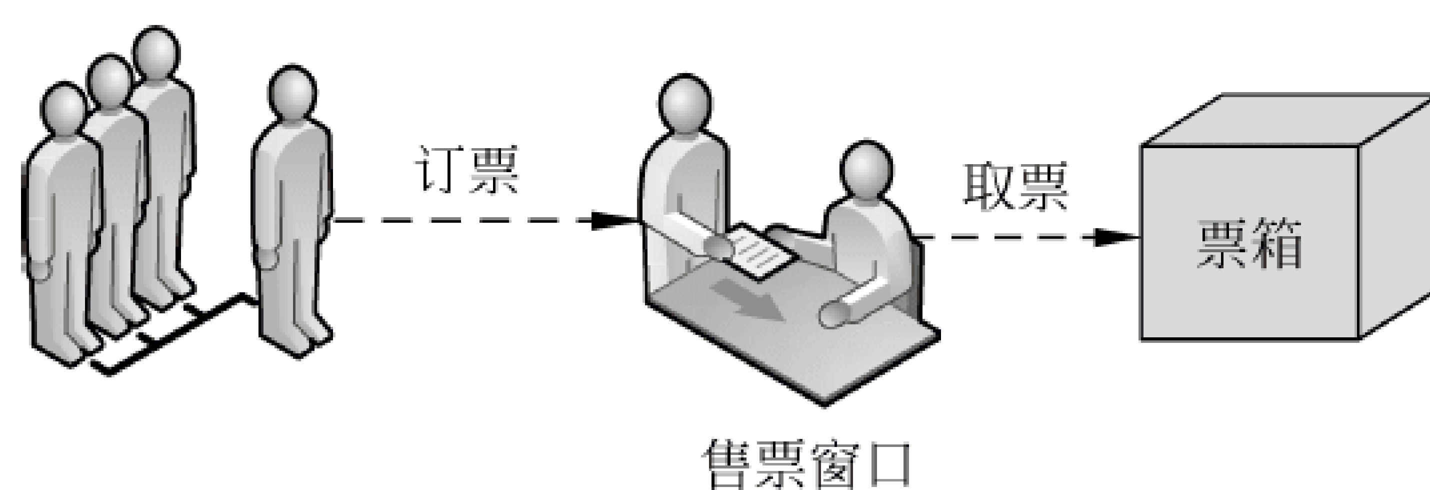


图 8-7　单窗口售票场景

(1) **售票准备**。向客户询问出发日期、目的地等购票信息。

(2) **售票**。查看票箱的剩余票数，如果有余票，则显示售票成功，并将剩余票数减 1；否则显示无票，售票失败。

使用面向对象程序设计方法，为票箱、售票窗口建立数据模型，并使用 Java 语言将它们定义成类，然后编写主方法 main()实现具体的售票服务流程。例 8-5 给出了完整的 Java 售票服务演示程序。这是一个单线程(即只有一个主线程)串行程序，模拟通过一个售票窗口向 5 位客户提供售票服务。

例 8-5　一个单线程串行的 Java 售票服务演示程序(JTicketST.java)

```
 1  public class JTicketST {                                  //主类:单线程串行的售票演示程序
 2      public static void main(String[] args) {              //主方法
 3          TicketBox    tb  =  new TicketBox(4);             //创建票箱,初始化有 4 张票
 4          TicketWindow tw  =  new TicketWindow(tb);         //创建一个售票窗口
 5          //模拟通过一个售票窗口向 5 位客户提供售票服务
 6          long sTime  =  System.currentTimeMillis();        //记录开始时间
 7          for (int n  =  1; n <= 5; n++)                    //循环提供 5 次售票服务
 8              tw.serviceAlgorithm();
 9          long eTime  =  System.currentTimeMillis();        //记录结束时间
10          System.out.println( "用时: "  + (eTime - sTime)/1000.0  + "秒" );
11  } }
12
13  class TicketBox {                                         //描述票箱的类 TicketBox
14      private int num  =  0;                                //剩余票数
15      public TicketBox(int x) {   num  =  x;   }            //构造方法
16      public int get() {   return num;   }                  //读取剩余票数
17      public void set(int x) {   num  =  x;   }             //设置剩余票数
18  }
19
20  class TicketWindow {                                      //提供售票服务的售票窗口类
21      private TicketBox tBox;                               //从票箱对象 tBox 中取票
22      public TicketWindow(TicketBox p)                      //构造方法
23      {   tBox  =  p;   }
24      public void prepare() {        //模拟售票前的一些准备工作,例如询问出发日期、目的地等
25          System.out.println(Thread.currentThread().getName()  + ": 购票前准备 ...");
26          try {
27              Thread.sleep(100);                  //休眠(暂停)0.1 秒,模拟购票前的准备工作
28          }
29          catch(InterruptedException e)           //捕捉 sleep()方法可能抛出的异常
```

```
30            {  System.out.println( e.getMessage() );   return;   }
31        }
32        public void sale() {                          //具体的售票算法
33            int tickets = tBox.get();                 //读取剩余票数
34            if (tickets > 0) {                        //如果有票
35                tickets --;                           //售出一张票,将剩余票数减 1
36                tBox.set(tickets);                    //设置票箱的剩余票数
37                System.out.println(Thread.currentThread().getName() +
38                                        ": 成功,剩余票数 " + tickets);
39            }
40            else  System.out.println(Thread.currentThread().getName() + ": 无票");  //无票
41        }
42        public void serviceAlgorithm() {              //描述完整售票服务流程的算法
43            prepare();                                //模拟售票前的准备工作
44            sale();                                   //模拟售票
45    }  }
```

```
Problems  Javadoc  Declaration  Console
<terminated> JTicketST [Java Application] C:\Java\jre
main: 购票前准备...
main: 成功，剩余票数 3
main: 购票前准备...
main: 成功，剩余票数 2
main: 购票前准备...
main: 成功，剩余票数 1
main: 购票前准备...
main: 成功，剩余票数 0
main: 购票前准备...
main: 无票
用时: 0.503秒
```

图 8-8 例 8-5 程序的运行结果

在 Eclipse 集成开发环境中运行例 8-5 的程序,运行结果如图 8-8 所示。

例 8-5 程序将票箱的初始票数设为 4,然后模拟向 5 位客户提供售票服务。图 8-8 依次显示了每位客户的购票过程信息,其中最后一位客户购票时显示“无票”。图 8-8 中的 main 表示主线程,即在主线程 main 中运行售票服务算法。只通过一个售票窗口向 5 位客户提供售票服务的模拟用时为 0.503 秒。

8.3.2 多线程售票服务演示程序

增加售票窗口可以有效提高售票服务能力,减少客户排队时间。售票服务程序可以引入多线程,并在多个线程中同时运行售票服务算法。这相当于是通过多个售票窗口同时售票,如图 8-9 所示。

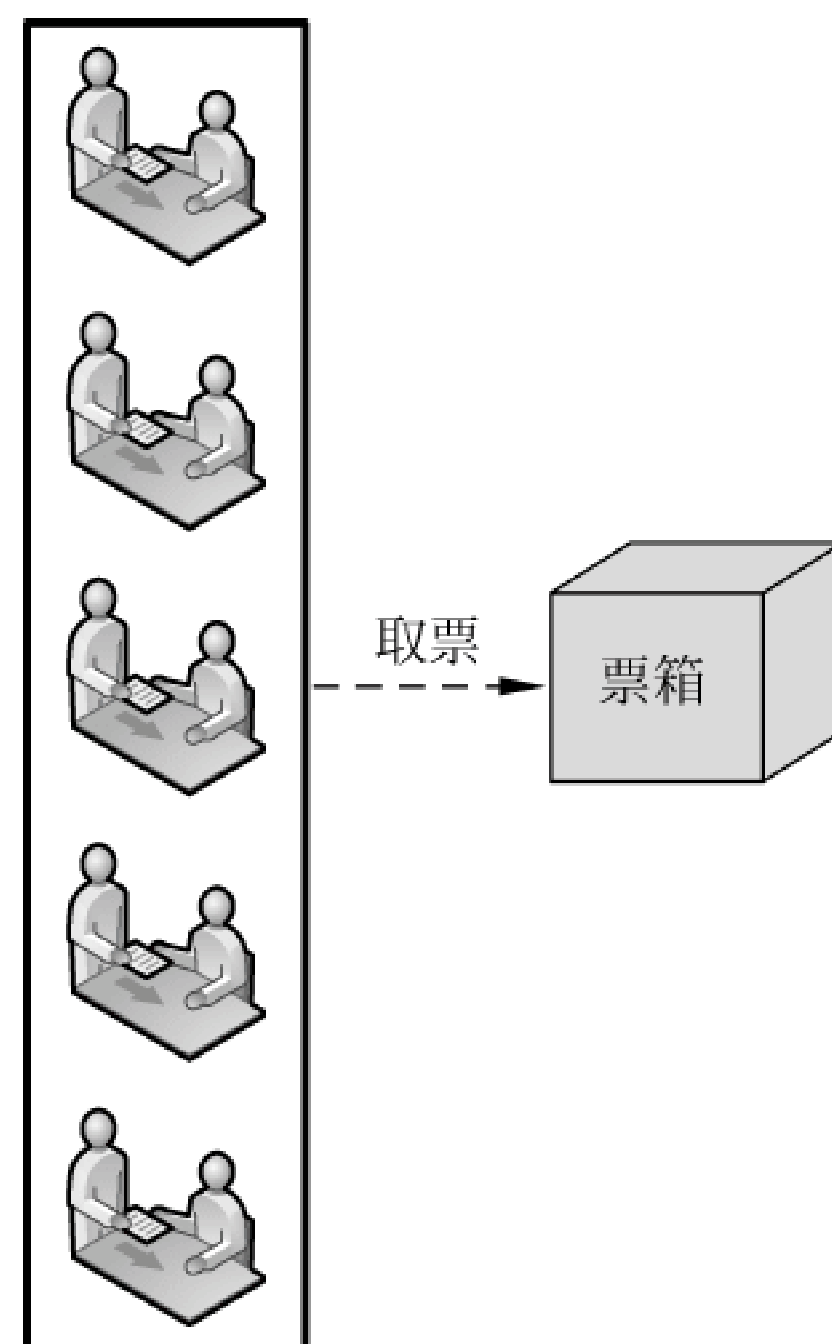


图 8-9 多窗口同时售票的场景

需要注意的是,虽然是多个售票窗口同时售票,但票箱只有一个。多个售票窗口需从同一票箱中取票,即共用一个票箱。对应到售票服务程序中,这意味着在多个线程中同时运行的售票服务算法,它们需要从同一票箱获取票务数据,即多个同时运行的售票服务算法会共用同一个票箱对象。

例 8-6 给出了一个多线程并发的 Java 售票服务演示程序。

例 8-6　一个多线程并发的 Java 售票服务演示程序(JTicketMT. java)

```
1  public class JTicketMT {                                  //主类:多线程并发的售票演示程序
2      public static void main(String[ ] args) {             //主方法
3          TicketBox    tb = new TicketBox(4);               //创建票箱,初始化有 4 张票
4          TicketWindow tw = new TicketWindow(tb);           //创建售票窗口(是一个算法对象)
5          //模拟 5 个售票窗口同时向 5 位客户提供售票服务
6          Thread t1 = new Thread(tw, "窗口 1");             //子线程 t1～t5 执行同样的售票算法 tw
7          Thread t2 = new Thread(tw, "窗口 2");   Thread t3 = new Thread(tw, "窗口 3");
8          Thread t4 = new Thread(tw, "窗口 4");   Thread t5 = new Thread(tw, "窗口 5");
9          //启动 5 个子线程,5 个售票窗口同时开始售票
10         long sTime = System.currentTimeMillis();         //记录开始时间
11         t1.start();  t2.start();  t3.start();  t4.start();  t5.start();
12         //主线程等待 5 个子线程都执行结束
13         while ( t1.getState() != Thread.State.TERMINATED ||
14             t2.getState() != Thread.State.TERMINATED ||
15             t3.getState() != Thread.State.TERMINATED ||
16             t4.getState() != Thread.State.TERMINATED ||
17             t5.getState() != Thread.State.TERMINATED )
18         { }                                               //空循环,等待 5 个子线程结束
19         long eTime = System.currentTimeMillis();         //记录结束时间
20         System.out.println( "用时: " + (eTime - sTime)/1000.0 + "秒" );
21 }  }
22
23 class TicketBox { … }                        //描述票箱的类 TicketBox,同例 8-5,省略代码
24
25 class TicketWindow {                         //提供售票服务的售票窗口类
26 class TicketWindow  implements Runnable {    //实现 Runnable 接口才能在线程中运行
27     ……                                       //此处同例 8-5 中的代码第 21～41 行,省略代码
28     public void serviceAlgorithm() {         //描述完整售票服务流程的算法
29     public void run() {          //必须实现 run()方法,描述可在线程中运行的售票服务算法
30         prepare();                           //模拟售票前的准备工作
31         sale();                              //模拟售票
32 }  }
```

在 Eclipse 集成开发环境中运行例 8-6 的程序,运行结果如图 8-10 所示。



图 8-10　例 8-6 程序的运行结果

例 8-6 程序将票箱的初始票数设为 4,然后在 5 个线程同时运行售票服务算法,模拟 5 个售票窗口同时向 5 位客户售票。图 8-10 依次显示了每个窗口的售票过程信息。因为线程的执行调度具有随机性,因此图 8-10 所显示的窗口售票次序看起来也是随机的。

这里通过例 8-6 对多线程再做如下两点说明。

(1) 多线程能提高程序效率。

例 8-6 使用 5 个线程模拟 5 个窗口同时售票,每个窗口只服务一位客户,整个售票过程

的模拟用时为 0.11 秒。而例 8-5 使用单线程模拟一个窗口服务 5 位客户，其模拟用时为 0.503 秒(见图 8-8)。可以看出，多线程并发程序比单线程串行程序具有更高的服务效率。

运用多线程并发编程技术可以充分利用 CPU 的计算能力，减少 CPU 的空闲等待时间，这样就能有效提高程序的服务效率。

(2) 多线程之间存在互斥操作。

再仔细研究一下图 8-10，可以发现它所显示的售票结果是错误的。例如，票箱总共只有 4 张票，按说只能售出 4 张票，售第 5 张票时应该显示"无票"。但图 8-10 中的窗口 5 在销售第 5 张票时不但售票成功，而且所显示的剩余票数为 3。这是为什么呢？我们回顾一下售票窗口的服务流程。

- **售票准备**。向客户询问出发日期、目的地等购票信息。多窗口售票时，各窗口在售票准备这个环节可以同时进行，互不干扰。换句话说，售票准备环节所做的操作是**可并发**的操作。
- **售票**。查看票箱的剩余票数，如果有余票，则显示售票成功，并将剩余票数减 1；否则显示无票，售票失败。多窗口售票时，因为多个窗口共用一个票箱，因此同一时刻只能有一个窗口从票箱中取票，否则会造成混乱。换句话说，售票环节所做的取票操作是**互斥**操作，不能在多个线程中重叠交叉进行。

多线程并发程序需要对类似"取票"这样的互斥操作做特殊处理，否则就会出现错误。例如，例 8-6 程序没有对"取票"这个互斥操作做任何特殊处理，运行时就会出现如图 8-10 所示的错误售票结果。

8.3.3 多线程中的互斥操作

多线程并发程序包含多个线程，每个线程运行一个算法。执行时各线程轮流切换，并发执行算法。如果两个线程中的算法不能重叠交叉执行，则这两个算法被称为是**互斥操作**。

1. 什么样的操作是互斥操作

一个程序可以划分成多个算法，将算法分散交由不同的线程去执行，这样可以实现多线程并发执行。不同线程之间可能需要共享数据，这样就可以实现多线程协同工作。

如果多个线程共享数据，则在不同线程中同时**访问共享数据**就可能是互斥操作。例如，8.3.2 节例 8-6 的多线程售票服务程序在 5 个线程中同时运行售票服务算法，它们从同一票箱获取票务数据。票务数据就是一个被多线程共享的数据。假设有两个线程(线程 1、线程 2)同时访问票箱里的票务数据，图 8-11 给出了 3 种不同的并发访问形式。

图 8-11(a)，**读-读**：两个线程都去读取票箱里的剩余票数。这种访问形式是两个线程在并发读取共享数据。

图 8-11(b)，**改-读**：线程 1 从票箱取一张票(即将票箱里的票数减 1)，而在取票过程中计算机切换到线程 2，线程 2 从票箱读取剩余票数。这种访问形式是一个线程在修改共享数据，而另一个线程在并发读取共享数据。

图 8-11(c)，**改-改**：线程 1 和线程 2 各从票箱里取出一张票，而且两个取票过程有重叠交叉。这种访问形式是两个线程在并发修改共享数据。

在线程中运行的算法，其修改共享数据的操作可细分为"**读取-修改-写回**"3 步。只有在

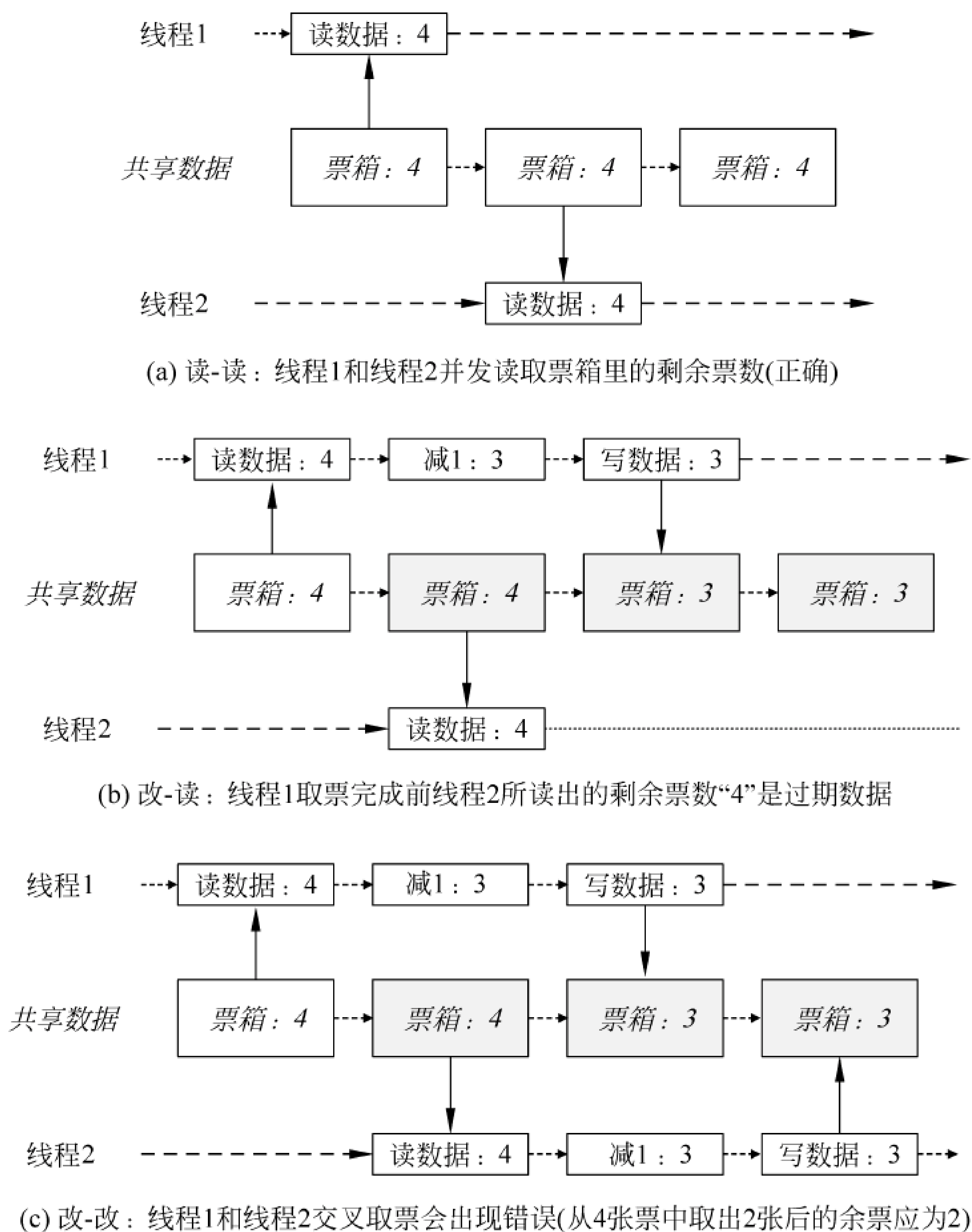


图 8-11　并发访问共享数据的 3 种形式

这 3 个步骤都执行完成后，修改过程才算结束。从图 8-11(b)、图 8-11(c)可以看出，当线程 1 修改共享数据的 3 个步骤全部完成之前，其他线程不能去并发读取或修改这个数据，否则就会出现错误。图 8-11(b)中的“**改-读**”操作是互斥操作，8-11(c)中的“**改-改**”操作也是互斥操作。并发执行互斥操作会导致程序出现错误的运行结果。

多线程并发程序需要对不同线程中运行的互斥操作做特殊处理，否则会产生错误的运行结果。产生错误的原因在于，本来不能重叠交叉执行的互斥操作会因线程切换而出现了重叠交叉。这种现象在单线程串行程序中不会发生，它是多线程并发程序所特有的问题。如何避免互斥操作的重叠交叉执行呢？Java 语言为此专门提供了一种**同步**(synchronization)机制。

2. 对互斥操作算法进行同步

多线程并发程序包含多个线程，每个线程运行一个算法。执行时各线程轮流切换，并发执行算法。如果线程中运行的算法是互斥操作，则需要使用 Java 语言中的同步机制对互斥操作算法进行同步。

所谓对互斥操作算法进行同步，就是使用关键字 **synchronized** 将互斥操作算法定义成一个**同步方法**(synchronized method)。程序执行时，一旦某个线程中的同步方法开始执行，所有其他线程中与该同步方法互斥的方法都不会被执行，直到同步方法所包含的指令序列执行结束，这就是 Java 语言里的同步机制。简单地说，同步方法所包含的指令序列必须一次性同步执行完成，其执行过程不会因线程切换而被其他线程里的互斥方法中途打断。

使用同步机制可以避免互斥操作的重叠交叉执行。图 8-11(b)中的"改-读"操作和图 8-11(c)中的"改-改"操作都属于互斥操作。通过 Java 语言的同步机制可以控制这些互斥操作只能按照图 8-12 所示的并发形式执行，不会出现重叠交叉。

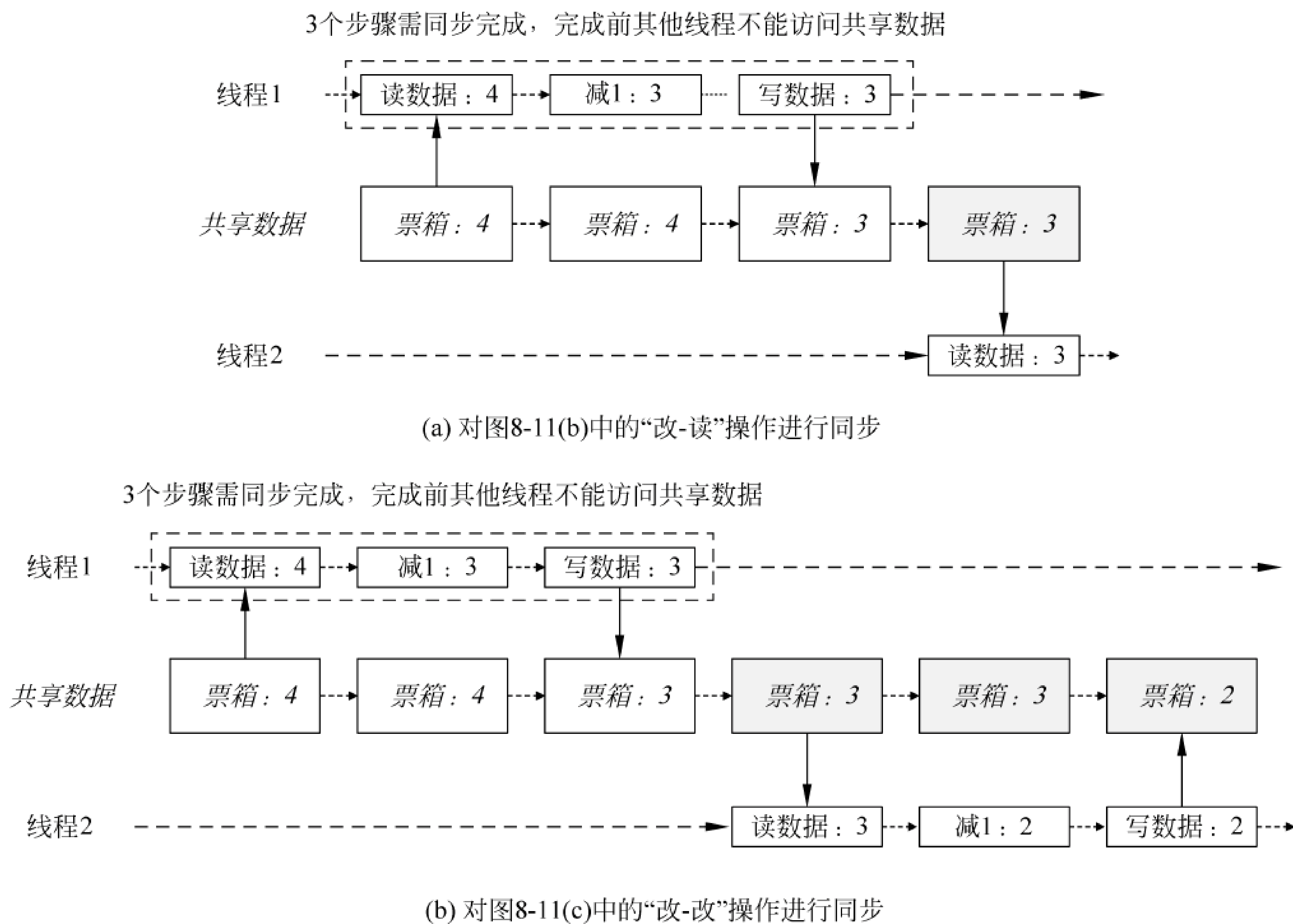


图 8-12 对图 8-11 中的互斥操作进行同步

Java 语言将互斥操作算法定义成同步方法的语法格式如下：

```
synchronized 方法类型 方法名(形式参数列表 ) {
    …     //此处编写互斥操作的算法代码
}
```

3. 为多线程售票服务程序添加同步机制

8.3.2 节例 8-6 的多线程售票服务程序在 5 个线程中同时运行售票服务算法，模拟 5 个窗口同时售票。售票窗口的服务流程包括售票准备和售票两个环节，其中的售票准备环节是可以并发的操作，而售票环节是互斥的操作。对售票环节所做的互斥操作需要进行同步，否则就会得到错误的售票结果(见图 8-10)。

修改例 8-6 的程序，对售票窗口类 TicketWindow 中的售票方法 sale()进行同步，即定义售票方法 sale()时增加关键字 synchronized，将其定义为同步方法。在对售票方法 sale()进行同步之后，再次执行程序就能得到正确的售票结果，如图 8-13 所示。



图 8-13　将 8.3.2 节例 8-6 程序中售票方法 sale()同步之后的售票结果

为便于后续讲解，例 8-7 给出修改后完整的售票窗口类 TicketWindow 定义代码。

例 8-7　对售票方法 sale()进行同步的售票窗口类 TicketWindow

```
 1  class TicketWindow  implements Runnable {  //可运行的售票窗口类 TicketWindow
 2      private TicketBox tBox;                //从票箱 tBox 取票
 3      public TicketWindow(TicketBox p)       //构造方法
 4      {  tBox = p;  }
 5      public void prepare() {      //模拟售票前的一些准备工作,例如询问出发日期、目的地等
 6          System.out.println(Thread.currentThread().getName() + ": 购票前准备 ...");
 7          try {
 8              Thread.sleep(100);             //休眠(暂停)0.1 秒,模拟购票前的准备工作
 9          }
10          catch(InterruptedException e)      //捕捉 sleep()方法可能抛出的异常
11          {  System.out.println( e.getMessage() );  return;  }
12      }
13      //public  void sale() {                //同步之前的售票方法
14      public synchronized  void sale() {     //同步之后的售票方法
15          int tickets = tBox.get();          //读取剩余票数
16          if (tickets > 0) {                 //如果有票
17              tickets --;                    //售出一张票,将剩余票数减 1
18              tBox.set(tickets);             //设置票箱的剩余票数
19              System.out.println(Thread.currentThread().getName() +
20                                          ": 成功,剩余票数 + tickets);
21          }
22          else  System.out.println(Thread.currentThread().getName() + ": 无票");
                                               //无票
23      }
24      public  void run() {                   //描述可在线程中运行的售票窗口算法 run()
25          prepare();                         //模拟售票前的准备工作
26          sale();                            //模拟售票
27      }
28  }
```

8.3.4 Java 同步机制的实现原理

Java 语言通过同步机制来控制多线程并发执行互斥操作时不会出现重叠交叉。在计算机内部,同步机制是如何实现的呢? Java 语言是通过对**当前对象**进行**加锁-解锁**来实现同步机制的,如图 8-14 所示。

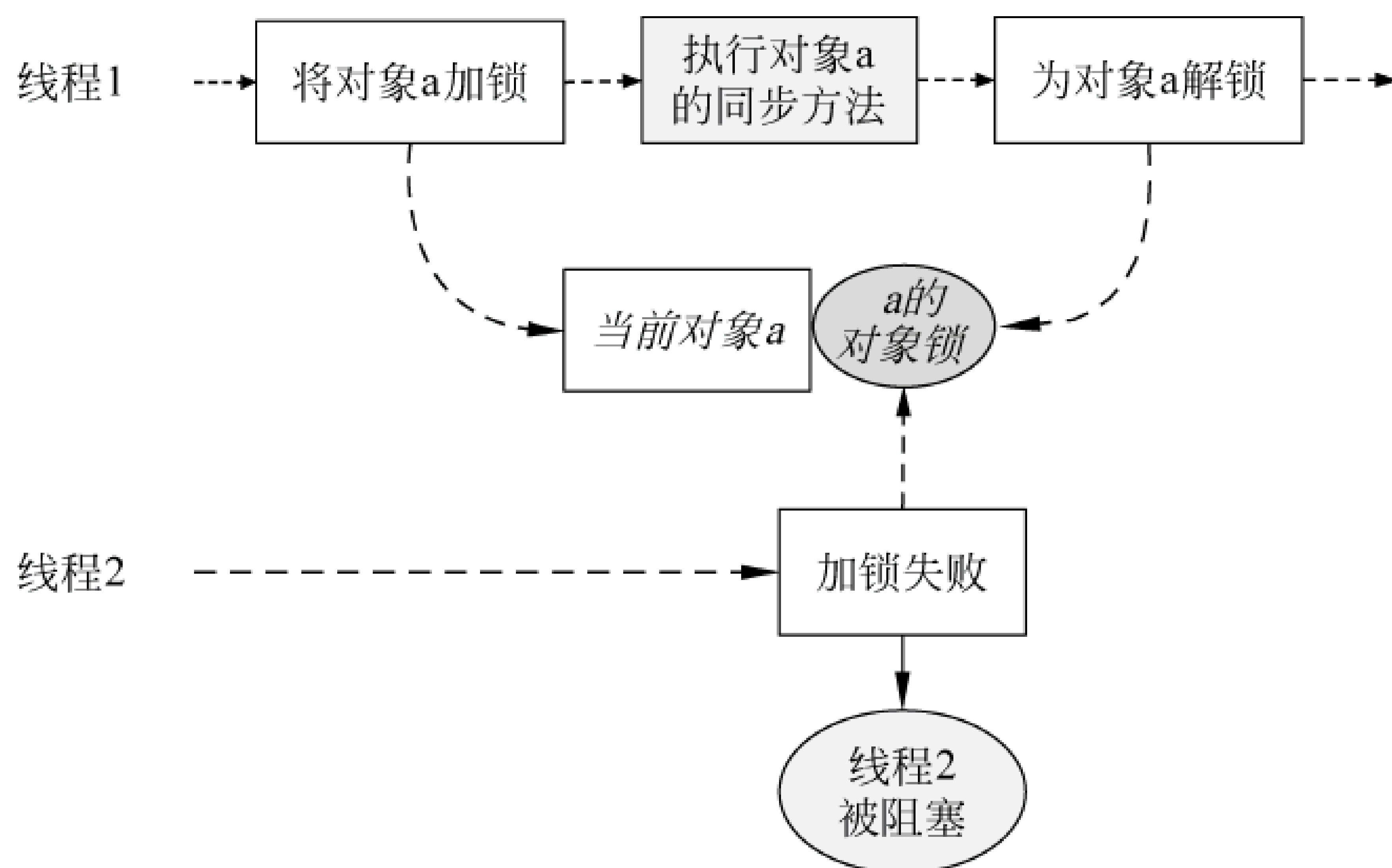


图 8-14 通过对当前对象进行加锁-解锁来实现同步机制

Java 虚拟机在线程中执行某个对象 a 的同步方法时,会首先将对象 a(称作**当前对象**)**加锁**。如果加锁成功(例如图 8-14 中的线程 1),则执行同步方法,执行结束后再将对象解锁。如果当前对象已被别的线程加锁,则加锁失败(例如图 8-14 中的线程 2),当前线程进入阻塞状态,直到当前对象被解锁。下面对 Java 同步机制的"**加锁-解锁**"原理做详细说明。

1. 什么是对象锁

Java 虚拟机为程序中的每个对象都自动设立一个**对象锁**。取得对象的锁,就是将对象**加锁**(lock),获得对象的拥有权;释放对象的锁,就是为对象**解锁**(unlock),取消已获得的拥有权。对象锁具有排他性,一个对象在同一时刻只能有一个线程拥有其对象锁。Java 语言也将对象锁称作**内部锁**(intrinsic lock)、**监视器锁**(monitor lock),或直接称作对象的**监视器**(monitor)。

2. 被锁定的当前对象是哪个对象

图 8-14 通过对当前对象进行加锁-解锁来实现同步机制,其中的"当前对象"指的是哪个对象呢?

- 如果线程所运行的同步方法属于对象 a,则对象 a 就是被锁定的当前对象。
- 如果两个线程中运行的同步方法都属于同一个对象,则两个线程锁定的就是同一个对象。这时两个线程将会为取得同一对象的对象锁而产生竞争,因为对象锁具有排他性。先取得对象锁的线程先运行,未取得对象锁的线程将被挂起,进入阻塞状态。Java 同步机制正是利用对象锁的排他性来控制多线程中的互斥操作只能串行执行,不会出现重叠交叉。

- 如果两个线程运行的同步方法属于不同对象，则两个线程锁定的就是不同对象。因为锁定的是不同对象，因此这两个线程在运行时不会产生竞争，互不干扰。

3. 同步语句

对互斥操作算法进行同步，可以使用关键字 synchronized 将互斥操作算法定义成一个同步方法。Java 语言还提供了另一种对互斥操作算法进行同步的语法形式，即**同步语句**(synchronized statement)。同步语句可以明确指定对哪个对象加锁，具体的语法格式如下：

```
synchronized(对象名) {
    …  //此处编写互斥操作的算法代码
}
```

例如，多线程售票服务程序中的售票算法是一个需要同步的互斥操作算法，例 8-7 代码第 14～23 行将其定义成售票窗口类 TicketWindow 中的一个同步方法 **sale**()。可以使用同步语句改写这个同步方法，修改后的示例代码如下：

```
//public synchronized  void sale() {          //将售票算法定义成一个同步方法
public  void sale() {                          //改用同步语句对售票算法进行同步
    synchronized(this ) {                      //对当前对象(即该方法所属的售票窗口对象)加锁
        int tickets = tBox.get();              //在大括号中编写需要同步的售票算法代码
        if (tickets > 0) {                     //有票有票
            tickets --;                        //售出一张票,将剩余票数减 1
            tBox.set(tickets);                 //设置票箱的剩余票数
            System.out.println(Thread.currentThread().getName() + ": 成功,剩余票数 " +
                               tickets);
        }
        else  System.out.println(Thread.currentThread().getName() + ": 无票");  //无票
    }
}
```

改写后的售票方法 sale()与例 8-7 中的同步方法 sale()完全等价。需要说明的是，示例代码中的同步语句使用关键字 **this** 来锁定当前对象，这个当前对象就是 **sale**()方法所属的售票窗口对象 tw(参见 8.3.2 节例 8-6 中的主方法代码)。当执行方法 sale()中的同步语句时，Java 虚拟机会锁定售票窗口对象 tw。

同步语句在使用上比同步方法更加灵活、方便，具体表现在以下两个方面。

(1) **同步语句可锁定任何对象**。同步方法只能通过锁定当前对象来实现同步。理论上，同步语句可以锁定任何对象来实现同步。如果多线程共享数据，同步语句一般是通过锁定被共享的数据对象来实现同步的。

(2) **同步语句可实现更细粒度的并发控制**。同步方法所同步的是整个方法，即方法中的所有指令，而同步语句可以只对方法中的部分指令进行同步。如果方法中只有部分代码是多线程互斥的，则只需要对这部分代码进行加锁、同步，这样可以提高程序的并发效率。使用同步语句，程序员可以让同一方法中的部分代码并发执行，部分代码串行执行，这样就能实现更细粒度的并发控制。

4. 对共享数据加锁

数据虽然能被多线程共享，但往往不能被同时操作，这是多线程互斥操作产生的主要原

因。使用同步语句直接锁定被共享的数据对象,这样可以对处理数据的算法进行同步,避免多线程同时操作共享数据。

例如在多线程售票服务程序中,售票算法之所以互斥是因为多个窗口共用一个票箱,同一时刻只能有一个窗口从票箱中取票。使用同步语句直接锁定票箱对象 tBox,这样就能对售票算法进行同步。修改前面示例代码中的售票方法 sale(),将对当前对象 **this** 加锁改为对票箱对象 **tBox** 加锁。修改后的示例代码如下:

```
public  void sale() {              //使用同步语句对售票算法进行同步
    //synchronized(this ) {        //修改前:对当前对象(即调用该方法的售票窗口对象)加锁
    synchronized(tBox ) {          //修改后:直接锁定票箱对象 tBox,对售票算法进行同步
        …                          //售票算法从 tBox 中取票。代码保持不变,此处省略
    }
}
```

直接锁定共享数据,然后对处理算法进行同步,这样所编写出的程序在逻辑上更加清晰,易于理解。图 8-15 给出了通过锁定票箱对象来同步多线程售票算法的运行示意图。其中线程 1 加锁成功,而线程 2 被阻塞,直到线程 1 解锁。

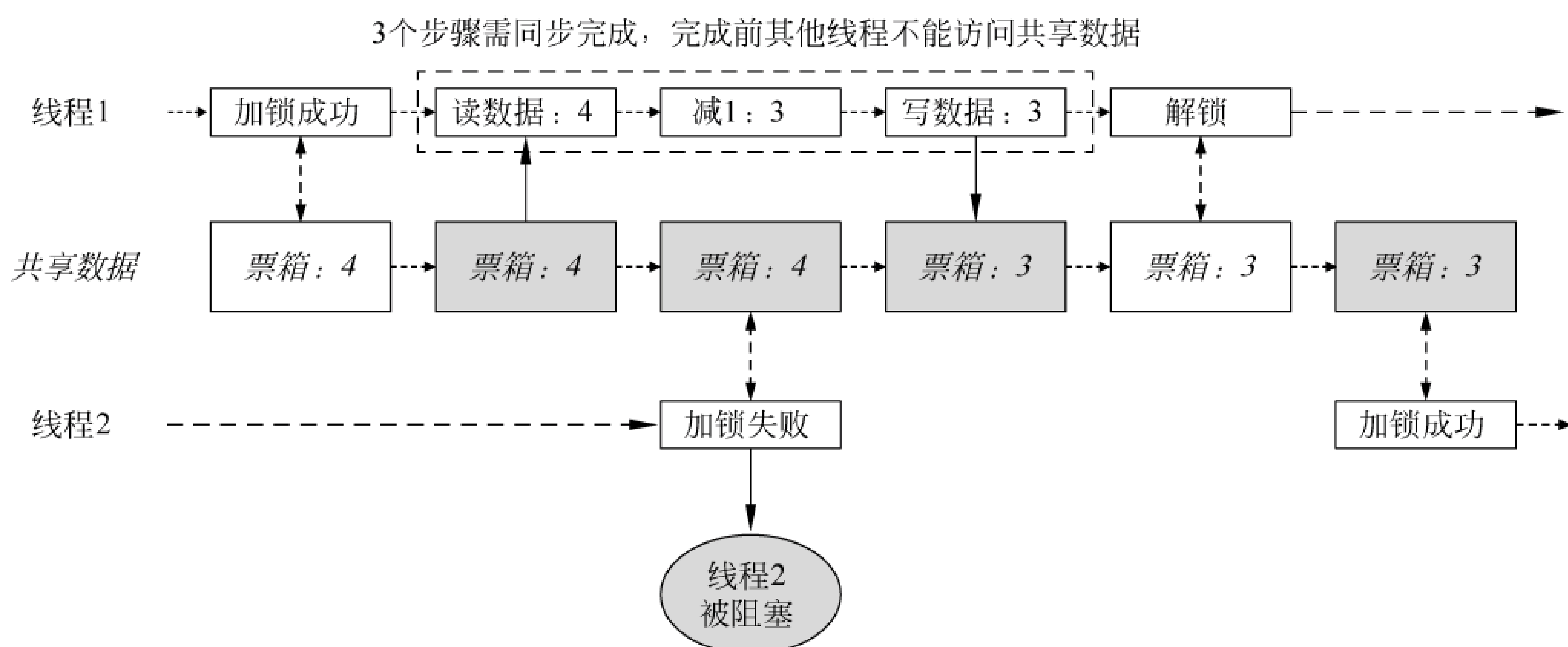


图 8-15 通过锁定票箱对象来同步多线程售票算法的运行示意图

本节最后给出一个典型的 Java 多线程并发数据处理程序代码框架(见例 8-8),以便读者从整体上掌握 Java 多线程并发程序的编程方法。

例 8-8 一个典型的 Java 多线程并发数据处理程序代码框架

```
1   class Data1 { … }                                   //描述待处理数据 1 的类
2   class Data2 { … }                                   //描述待处理数据 2 的类
3
4   class Algorithm  implements Runnable {              //描述数据处理算法的算法类
5       private Data1  d1;                              //指向待处理的数据 1
6       private Data2  d2;                              //指向待处理的数据 2
7       private void Algorithm(Data1 p1, Data2 p2) {    //构造方法
8       {  d1  =  p1;   d2  =  p2;   }                  //初始化待处理的数据
9       private void process1() {                       //数据 1 的处理算法
10          synchronized( d1 )                          //对处理数据 1 的算法代码进行同步
```

```
11            {  …  }              //具体的算法代码
12        }
13        private void process2() {  //数据 2 的处理算法
14            synchronized( d2 )     //对处理数据 2 的算法代码进行同步
15            {  …  }              //具体的算法代码
16        }
17        public void run() {        //实现 run()方法,描述可在线程中运行的数据处理算法
18            process1();            //处理数据 1
19            process2();            //处理数据 2
20        }
21    }
22
23    public class JDataProcessingMT {                    //多线程并发数据处理程序的主类
24        public static void main(String[] args) {        //主方法
25            Data1 dObj1 = new Data1();                  //创建待处理的数据对象 1
26            Data2 dObj2 = new Data2();                  //创建待处理的数据对象 2
27            Algorithm a = new Algorithm(dObj1, dObj2); //创建算法对象 a
28            //创建多个子线程,同时执行算法对象 a,实现多线程并发数据处理
29            Thread t1 = new Thread(a); Thread t2 = new Thread(a);  …
30            t1.start();  t2.start();  ……             //启动子线程,开始并发执行
31    }  }
```

本节习题

1. 下列关于多线程互斥操作的描述中,错误的是(　　)。
   A. 如果两个线程中的算法不能重叠交叉执行,则这两个算法被称为是互斥操作
   B. 修改内存对象中的数据,其修改过程可细分为“读取-修改-写回”3 步
   C. 如果多个线程共享数据,则在不同线程中同时修改共享数据就是互斥操作
   D. 如果多个线程共享数据,则在不同线程中同时读取共享数据就是互斥操作
2. 下列关于多线程互斥操作的描述中,错误的是(　　)。
   A. 多线程并发程序出现互斥操作重叠交叉执行的现象是因线程切换引起的
   B. 在单线程串行程序中也存在互斥操作重叠交叉执行的现象
   C. Java 虚拟机不能自动避免两个线程中的互斥算法重叠交叉执行
   D. 必须使用 Java 同步机制才能避免两个线程中的互斥算法重叠交叉执行
3. 下列关于同步方法的描述中,错误的是(　　)。
   A. 定义同步方法需使用关键字 synchronized
   B. 同步方法不会与其他线程里的互斥操作重叠交叉执行
   C. 不同线程中运行的同步方法修改同一个对象数据不会导致错误的运行结果
   D. 不同线程中运行的同步方法修改同一个对象数据可能会导致错误的运行结果
4. 下列关于 Java 同步机制“加锁-解锁”的描述中,错误的是(　　)。
   A. Java 虚拟机为程序中的每个对象都自动设立一个对象锁
   B. 一个对象在同一时刻只能有一个线程拥有其对象锁

C. 在线程中执行某个对象的同步方法必须首先取得该对象的对象锁

D. Java 语言通过调用对象的 getLock()方法取得该对象的对象锁

5. 下列关于同步语句的描述中,错误的是(　　)。

A. 使用同步语句可以指定对哪个对象加锁

B. 同步语句“synchronized(this){...}”表示对当前对象加锁

C. 同步语句只能锁定当前对象

D. 同步语句可实现更细粒度的并发控制

8.4 多线程之间的协同

假设有这样一个数据“**采集-处理**”系统,**处理模块**等待采集模块提交数据,如果发现有数据则取出处理,否则进入等待状态;**采集模块**检查自己提交的数据是否已被处理,如果已处理则提交下一个数据,否则也进入等待状态。类似的系统还有业务“**申请-审批**”系统、订单“**下单-处理**”系统等。可以将这类系统抽象成一种“**生产者-消费者**”(producer-consumer)数据处理模式,图 8-16 给出了一种最简单的“生产者-消费者”模式示意图。

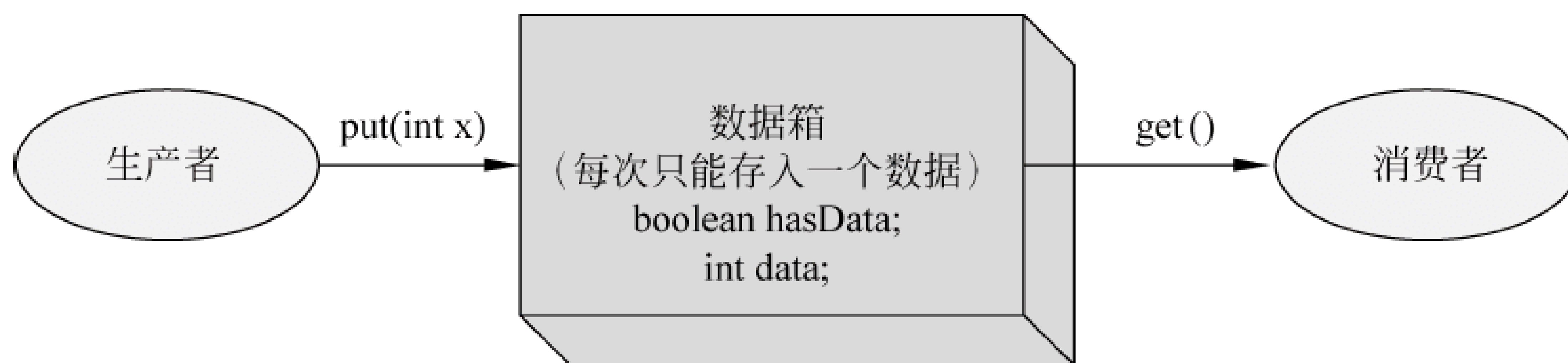


图 8-16　“**生产者-消费者**”模式示意图

在图 8-16 中,**生产者**模块不断生产数据并将其**存入**(put)数据箱;**消费者**模块持续监测数据箱,如果发现有数据则**取出**(get)处理。生产者和消费者共享**数据箱**,两者通过数据箱中转数据。可以采用多线程技术编写一个“生产者-消费者”模式数据处理程序,将生产者的**数据生产算法**和消费者的**数据处理算法**分别放入不同线程中运行,这就是一个多线程“生产者-消费者”模式数据处理程序,如图 8-17 所示。

多线程“生产者-消费者”模式数据处理程序需要满足如下两个约束条件。

(1) **对数据箱的操作需要同步**。生产者和消费者共用数据箱,对数据箱的操作是互斥操作,需要同步。在图 8-17 所示的生产者和消费者线程中,生产数据和处理数据这两个环节可以并发运行,但向数据箱存数据和从数据箱取数据这两个环节需要同步。

(2) **对数据箱的操作顺序是“先存后取,取完再存”**。只有当数据箱中有数据时,消费者才能取出数据;只有在数据箱中的数据已被消费者取出处理后,生产者才能放入下一个数据。在图 8-17 中,生产者线程和消费者线程应当交替运行。简单地说,对数据箱的操作逻辑是“先存后取,取完再存”。

编写多线程“生产者-消费者”模式数据处理程序,除了需要同步线程间的互斥操作之外,还需要控制各线程的运行次序,这就是多线程之间的**协同**(coordination)。Java 语言在同步机制的基础上又增加了一种**等待-唤醒**(wait-notify)机制,综合运用这两种机制才能实

现线程间的协同。

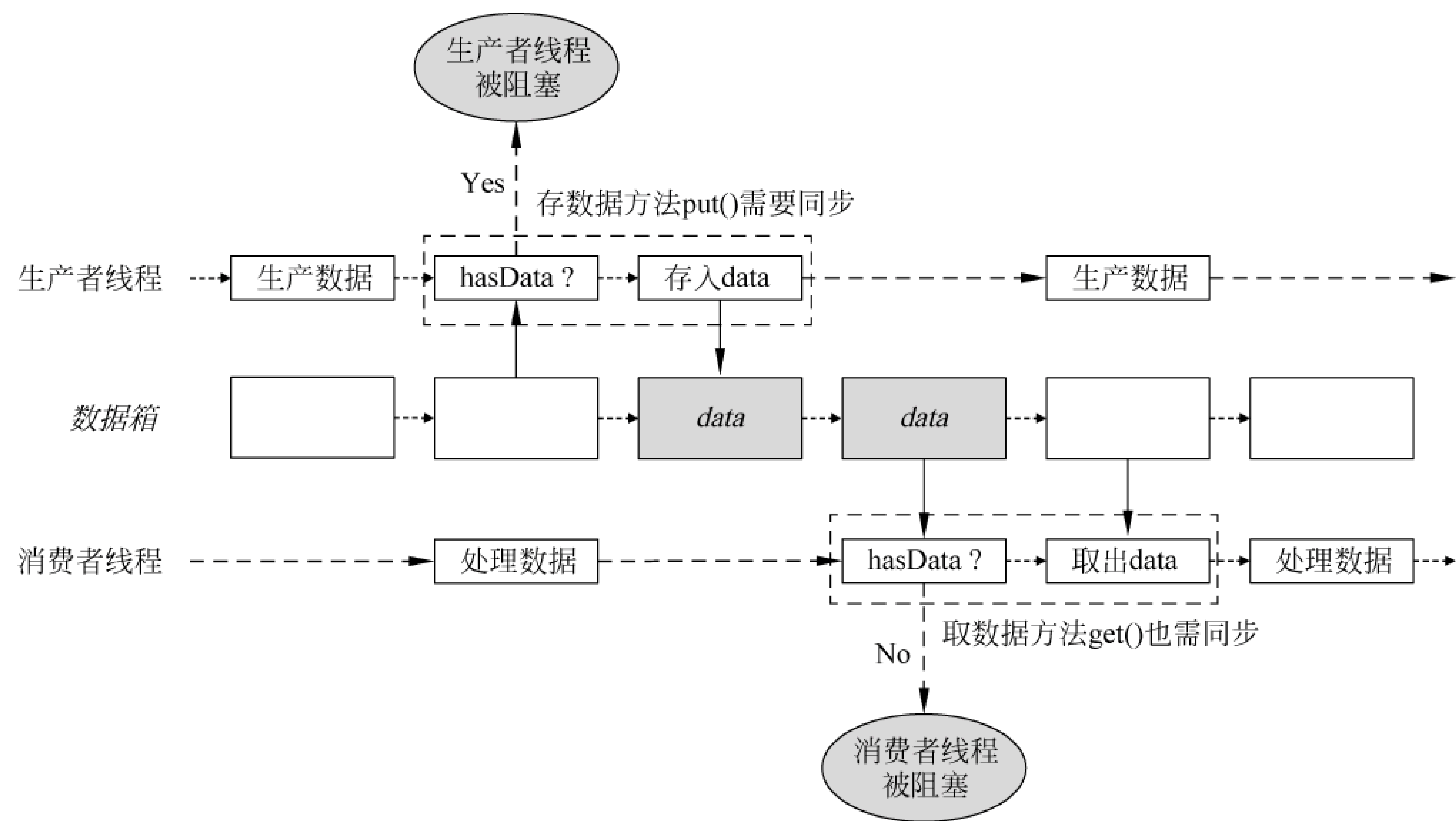


图 8-17　多线程**“生产者-消费者”**模式数据处理程序运行示意图

8.4.1　守护代码块

在图 8-17 中，只有当数据箱中有数据时，消费者才能去取数据。在设计取数据方法 get()时，需首先检查“数据箱中有数据”这个条件是否成立。如果条件不成立则应持续检查直至条件成立，待条件成立后再取出数据箱中的数据。取数据方法 get()应呈现如下算法代码结构：

```
int get() {                          //从数据箱取数据的方法
    while (hasData != true) { }      //循环检查条件"数据箱中有数据",直至条件成立
    return data;                     //条件成立后,取出并返回数据箱中的数据 data
}
```

持续检查算法条件直至条件成立，然后再进行后续处理，这样的算法代码块称为**守护代码块**(guarded block，或称为警戒代码块)。同理，向数据箱存数据方法 put()的算法代码也是一个守护代码块结构。

对图 8-16 中的数据箱进行抽象，将其定义成一个数据箱类 DataBox。生产者和消费者共用数据箱，对数据箱的操作是互斥操作，需要同步。例 8-9 给出一个具有同步机制的数据箱类 DataBox 示例代码。

例 8-9　一个具有同步机制的数据箱类 DataBox 示例代码(DataBox.java)

```
1  class DataBox {                             //存放共享数据的数据箱类
2      private boolean hasData = false;        //数据标记:标记数据箱里的数据是否有效
3      private int data = 0;                   //数据
```

```
4
5      public synchronized void put(int x) {  //生产者调用该方法,存入数据 x
6          while (hasData == true) { }        //如已有数据则空转等待,直到消费者将它取走
7          data = x;   hasData = true;        //存入数据 x,设置数据标记为 true(有效)
8      }
9
10     public synchronized int get() {        //消费者调用该方法,取出数据
11         while (hasData != true) { }        //如没有数据,则空转等待,直到生产者存入数据
12         int x = data;   hasData = false;  //取出数据,设置数据标记为 false(无效)
13         return x;                          //返回所取出的数据
14     }
15  }
```

结合图 8-17 中的多线程“生产者-消费者”模式数据处理程序运行示意图,对例 8-9 的数据箱类 DataBox 做以下分析。

(1) 数据标记 hasData。

数据箱类 DataBox 专门定义了一个布尔型字段 hasData,用于标记数据箱类里的数据 data 是否有效。

生产者线程调用方法 **put**()向数据箱存数据。如果 hasData 为 true 则说明上次存入的数据 data 还未被取走,不能再向 data 存入数据,必须等待消费者线程取走数据。

消费者线程调用方法 **get**()从数据箱取数据。如果 hasData 不为 true(即 false)则说明数据 data 已被取过一次,不能再重复读取,必须等待生产者线程存入新的数据。

(2) 同步方法中的空转等待。

例 8-9 中的方法 put()和 get()被定义成同步方法,因为它们是互斥操作。这两个同步方法在执行时都需要锁定数据箱对象。

假设数据箱对象中已经存放了一个数据,这时生产者线程执行方法 put()会通过 while 循环“**空转等待**”。空转的目的是等消费者线程执行方法 get()取走上次存入的数据,这样才能继续存入下一个数据。**此处注意**,消费者线程会因数据箱对象已被生产者线程锁定而阻塞,无法执行同步方法 get()。这样,程序执行出现僵持,方法 put()陷入了无休止的空转等待。同理,消费者线程在执行方法 get()时也可能会陷入无休止的空转等待。这种因加锁机制而导致的线程间互相阻塞现象称为**死锁**(deadlock)。在使用程序的用户看来,死锁就是程序长时间没有反应(俗称死机)。

例 8-9 所示的数据箱类 DataBox 之所以会产生死锁,是因为它在同步方法中使用了空转等待。线程执行同步方法会锁定数据箱对象,空转等待不会释放对象锁。当生产者线程和消费者线程中的任何一方拿到对象锁并空转等待时,另一方就会因拿不到对象锁而无法执行,这就造成了死锁。为了解决空转等待导致死锁的问题,Java 语言设计了一种新的等待方式,它被称为**阻塞等待**。

8.4.2 Java 等待-唤醒机制

在同步方法中使用“**空转等待**”会导致死锁。为了解决这个问题,Java 语言提供了一种新的等待方法,即**阻塞等待**。在同步方法中使用阻塞等待,当前线程会释放对象锁,然后暂

停执行并进入阻塞状态,CPU 转去执行其他线程;被阻塞的线程需要由其他线程唤醒,唤醒后再继续执行,这就是 Java 语言的“**等待-唤醒**”机制。

1. 阻塞等待 wait 和阻塞唤醒 notifyAll

Java 语言在根类 Object 中就定义了阻塞等待和唤醒的方法,它们分别是 **wait**()和 **notifyAll**(),参见 5.4 节。所有 Java 类都直接或间接继承了这两个方法。

(1) **阻塞等待方法** wait()。在同步方法或同步语句中调用 wait()方法,当前线程将释放对象锁并进入阻塞状态,等待唤醒。唤醒后再继续执行同步方法或同步语句中剩余的指令。每个处于阻塞状态的线程都对应一个阻塞它的对象锁。换句话说,每个处于阻塞状态的线程都是在等待某个对象的对象锁。方法 wait()只能在同步方法或同步语句中调用,这意味着调用 wait()方法时,当前线程一定占用着某个对象的对象锁。调用 wait()方法,当前线程将释放已取得的对象锁,然后被该对象锁阻塞。

(2) **阻塞唤醒方法** notifyAll()。方法 notifyAll()也只能在同步方法或同步语句中调用,这意味着执行 notifyAll()方法时,当前线程一定占用着某个对象的对象锁。调用 notifyAll()方法将唤醒所有被该对象锁阻塞的线程,然后继续当前线程的执行。被唤醒后的阻塞线程将转入就绪状态并在就绪队列中排队,等待 Java 虚拟机分配 CPU 控制权后再继续执行。**注**:根类 Object 中还定义了一个唤醒方法 notify(),该方法只会随机唤醒一个阻塞线程。

修改例 8-9 中数据箱类 DataBox 的 put()和 get()方法,将“空转等待”改成“等待-唤醒”,这样就能避免死锁。例 8-10 给出了修改后的数据箱类 DataBox 示例代码。

例 8-10　一个具有同步机制和等待-唤醒机制的数据箱类 DataBox 示例代码(DataBox.java)

```
 1  class DataBox {                                 //存放共享数据的数据箱类
 2      private boolean hasData = false;            //数据标记:标记数据箱里的数据是否有效
 3      private int data = 0;                       //数据
 4
 5      public synchronized void put(int x) {       //生产者调用该方法,存入数据 x
 6          while (hasData == true) {               //如已有数据则阻塞等待,直到消费者将它取走
 7              try {                   //方法 wait()可能会抛出勾选异常 InterruptedException
 8                  wait();             //当前的生产者线程被阻塞,等待被消费者线程唤醒
 9              }   catch(InterruptedException e) { }
10          }
11          data = x;   hasData = true;             //存入数据 x,设置数据标记为 true(有效)
12          notifyAll();                            //唤醒被阻塞的消费者线程,通知它们可以取数据了
13      }
14
15      public synchronized int get() {             //消费者调用该方法,取出数据
16          while (hasData != true) {               //如没有数据,则阻塞等待,直到生产者存入数据
17              try {                     //方法 wait()可能会抛出勾选异常 InterruptedException
18                  wait();                         //当前的消费者线程被阻塞,等待被生产者线程唤醒
19              }   catch(InterruptedException e) { }
20          }
```

```
21            int x = data; hasData = false; //取走数据,设置数据标记为 false(无效)
22            notifyAll();                   //唤醒被阻塞的生产者线程,通知它们可以生产数据了
23            return x;
24        }
25    }
```

这里对例 8-10 的数据箱类 DataBox 做如下两点说明。

(1) **阻塞等待 wait()可能会抛出勾选异常** InterruptedException。调用阻塞等待方法 wait()必须使用 try-catch 语句来捕获并处理这个勾选异常,否则编译不能通过。例如例 8-10 中的代码第 7～9 行和第 17～19 行。

(2) **阻塞等待-唤醒中 wait()和 notifyAll()的对应关系**。一个线程调用 wait()方法所造成的阻塞必须被另一个线程的 notifyAll()方法来唤醒,这两者之间具有严格的对应关系。例如,例 8-10 在 put()方法中调用 wait()方法(代码第 8 行)所造成的生产者线程阻塞需要被消费者线程调用 get()方法中的 notifyAll()方法(代码第 22 行)来唤醒,而 put()方法中调用 wait()方法(代码第 18 行)所造成的消费者线程阻塞则需要被生产者调用 put()方法中的 notifyAll()方法(代码第 12 行)来唤醒。

例 8-10 中的数据箱类 DataBox 存在两对阻塞-唤醒关系,比较复杂。请读者在阅读示例代码时,首先理解清楚数据箱"先存后取,取完再存"的操作逻辑,然后根据这个操作逻辑再分别解读"先存后取"中的阻塞-唤醒关系、"取完再存"中的阻塞-唤醒关系。

2. 线程安全类

例 8-10 所定义的数据箱类 DataBox 是一个存放数据的类,它综合运用了 Java 语言的同步机制和等待-唤醒机制。使用这个类可以在多线程并发程序中安全地存取共享数据,这样的类被称为**线程安全的**(thread safe)类。在多个线程中并发访问线程安全类的对象,访问时不需要再添加 Java 语言的同步机制或等待-唤醒机制。

实现相同功能的类,Java API 可能会有"线程安全"和"非线程安全"两个版本。例如:

1) 字符串类

- 线程安全版本:String、StringBuffer(可变字符串类)是线程安全的。
- 非线程安全版本:StringBuilder,这是从 JDK 1.5 开始提供的非线程安全的可变字符串类。

2) 集合类

- 线程安全版本:Vector、Hashtable 是 JDK 早期版本提供的线程安全的集合类。
- 非线程安全版本:ArrayList、LinkedList、HashSet、HashMap 等是从 JDK 1.2 开始提供的非线程安全的集合类。

实现相同功能的类,线程安全版本的运行效率比非线程安全版本要低,因为同步和等待-唤醒机制需要花费额外的系统开销。

JDK 早期版本提供的是线程安全的类。考虑到运行效率的问题,JDK 逐步调整思路,开始提供非线程安全的类。在开发单线程串行程序时应选用非线程安全版本的类,选用线程安全版本没有意义。而在开发多线程并发程序时应选用线程安全版本的类,否则程序员须自己添加同步机制和等待-唤醒机制。

8.4.3 多线程"生产者-消费者"模式编程

基于例 8-10 中定义的数据箱类 DataBox，例 8-11 给出了一个完整的多线程"生产者-消费者"模式数据处理程序的示例代码。其生产者线程和消费者线程交替运行的过程如图 8-17 所示。

例 8-11 一个多线程"生产者-消费者"模式数据处理程序的示例代码(JProducerConsumer.java)

```
 1  public class JProducerConsumer {        //多线程"生产者 - 消费者"模式数据处理程序的主类
 2      public static void main(String[] args) {  //主方法
 3          DataBox db = new DataBox();           //创建一个数据箱对象 db
 4          Producer p = new Producer( db );      //创建一个生产者对象 p
 5          Consumer c = new Consumer( db );      //创建一个消费者对象 c
 6          Thread tp = new Thread(p, "Producer"); //创建一个运行生产者对象 p 的线程 tp
 7          Thread tc = new Thread(c, "Consumer"); //创建一个运行消费者对象 c 的线程 tc
 8          tp.start();     tc.start();           //启动线程
 9  } }
10
11  class Producer implements Runnable {          //生产者算法类:生产数据,然后存入数据箱中
12      private DataBox dBox;                     //存放数据的数据箱(线程安全类)
13      //以下为模拟的原始数据,将被依次存入数据箱
14      private int[] x = { 1, 3, 5, 7, 9, -1 }; //-1:数据结束标志
15      public Producer(DataBox d)                //构造方法
16      {  dBox = d;  }
17      public void run() {                       //模拟生产的算法
18          for (int n = 0; n < x.length; n++) {
19              System.out.println( Thread.currentThread().getName() + ": " + x[n] );
20              dBox.put( x[n] );                 //模拟生产出一个数据,存入数据箱
21              try {
22                  Thread.sleep(100);            //休眠(暂停)0.1 秒,模拟复杂的数据生产过程
23              }  catch(InterruptedException e) { }
24          }
25  }}
26
27  class Consumer implements Runnable {          //消费者算法类:从数据箱取数据,然后处理
28      private DataBox dBox;                     //存放数据的数据箱,将从中取数据
29      public Consumer(DataBox d)                //构造方法
30      {  dBox = d;  }
31      public void run() {                       //模拟消费的算法
32          while (true) {
33              int x = dBox.get();               //从数据箱中取出一个数据,计算其平方值
34              if (x == -1) {                    //-1:数据结束标志
35                  System.out.println("\t" + Thread.currentThread().getName() + ": 数据结束" );
36                  return;
37              }
38              System.out.println("\t" + Thread.currentThread().getName() + ": " + x * x);
39              try {
40                  Thread.sleep(100);            //休眠(暂停)0.1 秒,模拟复杂的数据处理过程
41              }  catch(InterruptedException e) { }
42          }
43  } }
```

在例 8-11 中,生产者线程生产出原始数据 1、3、5、7、9,并逐个将其存入数据箱;消费者线程发现数据后立即取出数据,计算其平方值。生产者线程最后在数据箱中存入-1,这是一个结束标记,表示数据结束。在 Eclipse 集成开发环境中运行例 8-11 的程序,运行结果如图 8-18 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> JProducerConsumer [Java Application]
Producer: 1
        Consumer: 1
Producer: 3
        Consumer: 9
Producer: 5
        Consumer: 25
Producer: 7
        Consumer: 49
Producer: 9
        Consumer: 81
Producer: -1
        Consumer: 数据结束
```

图 8-18 例 8-11 程序的运行结果

本节习题

1. 下列关于多线程协同的描述中,错误的是(　　)。
   A. 多线程之间需要协同是因为它们之间存在互斥操作
   B. 多线程之间需要协同是因为它们既需要同步互斥操作,又需要控制运行次序
   C. 综合运用 Java 语言的同步机制和等待-唤醒机制才能实现线程间的协同
   D. 编写多线程"生产者-消费者"模式数据处理程序时需要多线程协同
2. 下列关于空转等待和死锁的描述中,错误的是(　　)。
   A. 算法空转等待是因为执行算法的条件还没有满足
   B. 在同步方法中使用空转等待可能会产生死锁
   C. 在用户看来,程序运行出现死锁就是程序长时间没有反应
   D. 在用户看来,程序运行出现死锁就是程序出现错误,中途退出
3. 阻塞等待方法 wait()是类(　　)定义的。
   A. Object　　B. System
   C. Thread　　D. 实现 Runnable 接口的算法类
4. 下列关于阻塞等待方法 wait()的描述中,错误的是(　　)。
   A. 阻塞等待方法 wait()只能在同步方法或同步语句中调用
   B. 调用 wait()方法,当前线程会释放对象锁
   C. 调用 wait()方法,当前线程会进入阻塞状态
   D. 调用 wait()方法,进入阻塞状态的当前线程会在休眠一定时间后自动恢复运行
5. 下列关于阻塞唤醒方法 notifyAll()的描述中,错误的是(　　)。
   A. 阻塞唤醒方法 notifyAll()只能在同步方法或同步语句中调用
   B. 调用 notifyAll()方法,会唤醒所有处于阻塞状态的线程

C. 调用 notifyAll()方法，会唤醒所有被当前线程所占用对象锁阻塞的线程

D. 执行 notifyAll()方法时，当前线程一定占用着某个对象的对象锁

6. 调用阻塞等待方法 wait()，调用错误的是(　　)。

A. 在同步方法中调用　　B. 在同步语句中调用

C. 在已取得对象锁的地方调用　　D. 在未取得对象锁的地方调用

7. 调用阻塞唤醒方法 notifyAll()，调用错误的是(　　)。

A. 在同步方法中调用　　B. 在同步语句中调用

C. 在已取得对象锁的地方调用　　D. 在未取得对象锁的地方调用

8. 下列关于线程安全类的描述中，错误的是(　　)。

A. 线程安全类运用了 Java 语言的同步机制

B. 线程安全类运用了 Java 语言的等待-唤醒机制

C. 多线程并发访问线程安全类的对象时需要添加 Java 同步机制

D. 多线程并发访问线程安全类的对象时不需要添加 Java 同步机制

8.5 定时执行的线程

本节先介绍一下 Java API 中的本地日期时间类 LocalDateTime，然后再讲解如何编写一个定时执行的程序。

8.5.1 本地日期时间类 LocalDateTime

Java API 提供了一个本地日期时间类 **LocalDateTime**，用于获取并存储计算机系统的本地时间(不含时区信息)。请读者阅读下面的本地日期时间类 LocalDateTime 说明文档。

java.time.**LocalDateTime** 类说明文档

public final class **LocalDateTime**
extends **Object**
implements **Temporal**, **TemporalAdjuster**, **ChronoLocalDateTime** < **LocalDate** >, **Serializable**

	修饰符	类成员(节选)	功能说明
1	static	LocalDateTime **now**()	获取表示本地当前时间的对象
2	static	LocalDateTime **of**(int year, int month, int dayOfMonth, int hour, int minute, int second)	创建一个本地时间对象
3	static	LocalDateTime **parse**(CharSequence text)	创建一个本地时间对象
4		int **getYear**()	获取年(月、日)
5		int **getHour**()	获取时(分、秒)
6		LocalDateTime **plusYears**(long years)	增加年(月、日)
7		LocalDateTime **plusHours**(long seconds)	增加时(分、秒)
8		LocalDateTime **minusYears**(long years)	减少年(月、日)
9		LocalDateTime **minusHours**(long seconds)	减少时(分、秒)
10		int **compareTo**(ChronoLocalDateTime <?> other)	比较时间的先后
		…	

例 8-12 给出一个本地日期时间类 LocalDateTime 的 Java 演示程序。

例 8-12 本地日期时间类 LocalDateTime 的 Java 演示程序(JLocalDateTimeTest.java)

```
1  import java.time.*;                              //导入 java.time 包中与时间相关的类
2  public class JLocalDateTimeTest {                //测试类:测试本地日期时间类 LocalDateTime
3      public static void main(String[] args) {         //主方法
4          LocalDateTime t = LocalDateTime.now();       //获取本地计算机系统的当前时间
5          System.out.println( t );                     //显示当前时间
6          System.out.println( "getYear():       " + t.getYear() );        //年
7          System.out.println( "getMonthValue(): " + t.getMonthValue() );  //月
8          System.out.println( "getDayOfMonth(): " + t.getDayOfMonth() );  //日
9          System.out.println( "getHour():       " + t.getHour() );        //时
10         System.out.println( "getMinute():     " + t.getMinute() );      //分
11         System.out.println( "getSecond():     " + t.getSecond() );      //秒
12 } }
```

8.5.2 定时执行的线程

为了方便程序员编写定时执行某些操作任务的程序,Java API 提供了两个类:一个是定时任务类 TimerTask;另一个是定时器类 Timer。

- **定时任务类 TimerTask**。定时任务类 TimerTask 实现了 Runnable 接口。它是一个算法类,用于描述定时执行的操作任务。
- **定时器类 Timer**。用于创建定时执行的守护线程(即后台线程),并在其中执行定时任务类 TimerTask 的算法对象。

例 8-13 给出一个定时显示本地系统时间的 Java 演示程序。

例 8-13 一个定时显示本地系统时间的 Java 演示程序(JTimerTest.java)

```
1  import java.util.*;            //导入 java.util 包中的类(其中包括类 TimerTask 和 Timer)
2  import java.time.*;            //导入 java.time 包中与时间相关的类
3
4  class ShowTime extends TimerTask {//继承 TimerTask 定义一个定时执行的程序任务类
5      public int count = 0;      //记录被定时执行的次数
6      public void run() {        //实现 run()方法,编写需要定时执行的算法代码
7          count++;
8          System.out.println( count + ": " + LocalDateTime.now() );  //显示系统时间
9  } }
10
11 public class JTimerTest {                       //测试类:定时执行某个程序任务
12     public static void main(String[] args) {    //主方法
13         Timer t = new Timer("Show time");       //创建一个定时器类 Timer 的对象 t
14         ShowTime st = new ShowTime();           //创建一个显示时间的定时任务对象 st
15         t.schedule(st, 0, 2000);    //创建后台线程并在其中每隔 2 秒执行一次任务对象 st
```

```
16          while (st.count < 5) {                    //检查定时执行的次数,执行 5 次后结束
17              try {
18                  Thread.sleep(1000);               //休眠 1 秒
19              } catch (InterruptedException e) { }
20          }
21          t.cancel();                               //结束(取消)定时任务
22          System.out.println( "TimerTask stopped" );
23  }  }
```

在 Eclipse 集成开发环境中运行例 8-13 的程序,运行结果如图 8-19 所示。

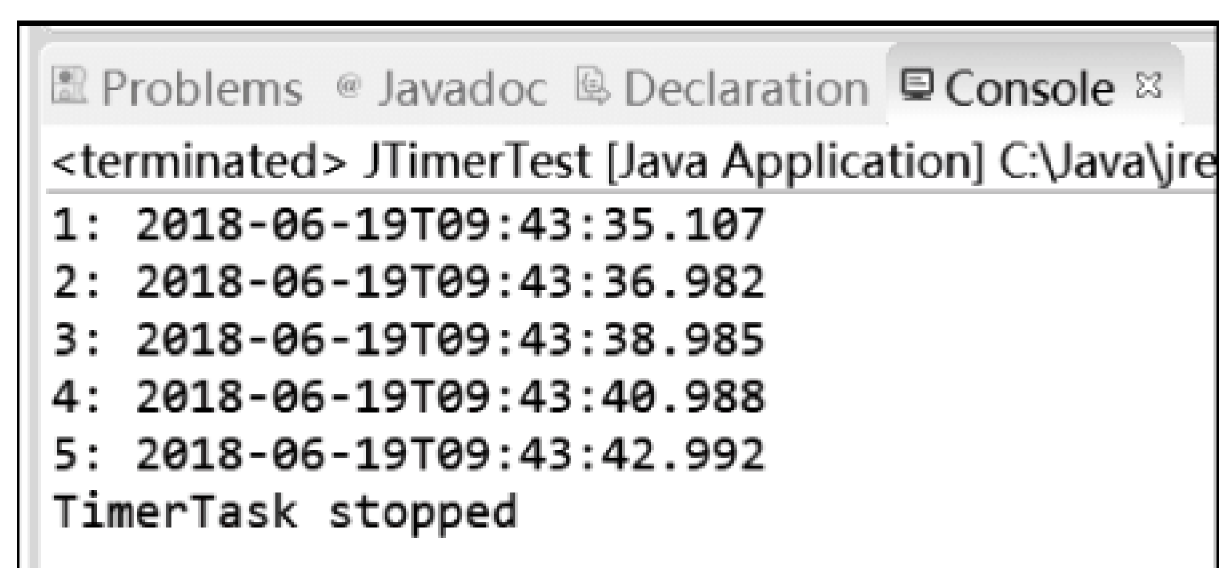


图 8-19　例 8-13 程序的运行结果

请读者阅读下面的定时任务类 TimerTask 和定时器类 Timer 说明文档。

java.util.**TimerTask** 类说明文档

public abstract class **TimerTask**
extends **Object**
implements **Runnable**

	修　饰　符	类成员(节选)	功 能 说 明
1	protected	**TimerTask**()	构造方法
2	abstract	void **run**()	定义需要定时执行的算法代码
3		void **cancel**()	取消定时任务

…

java.util.**Timer** 类说明文档

public class **Timer**
extends **Object**

	修　饰　符	类成员(节选)	功 能 说 明
1		**Timer**()	构造方法
2		**Timer**(String name)	构造方法
3		void **schedule**(TimerTask task, long delay, long period)	创建一个定时执行程序任务 task 的守护线程
4		void **cancel**()	终止定时线程

…

本节习题

1. 本地日期时间类 LocalDateTime 中获取本地当前时间对象的方法是(　　)。
   A. now()　　B. of()　　C. parse()　　D. getHour()
2. 本地日期时间类 LocalDateTime 中获取时间对象中月份的方法是(　　)。
   A. getYear()　　B. getMonthValue()
   C. getDayOfMonth()　　D. getHour()
3. 下列关于定时任务类 TimerTask 的描述中,错误的是(　　)。
   A. 定时任务类 TimerTask 实现了 Runnable 接口
   B. 定时任务类 TimerTask 是线程类 Thread 的子类
   C. 定义一个定时执行的程序任务类需继承类 TimerTask
   D. 继承类 TimerTask 需重写其中的 run()方法,编写定时执行的算法代码
4. 定时器类 Timer 中启动定时执行程序任务的方法是(　　)。
   A. schedule()　　B. cancel()　　C. run()　　D. start()
5. 定时器类 Timer 中结束定时执行程序任务的方法是(　　)。
   A. schedule()　　B. cancel()　　C. run()　　D. stop()

8.6 swing 框架中的线程

本节讲解图形用户界面程序开发框架 swing 中的线程,以及如何在线程中正确操作图形界面中的组件。

8.6.1 事件分发线程

开发图形用户界面程序需要用到 swing 框架,其中包含一组图形组件。swing 框架在内部会用到一个重要的线程,这就是**事件分发线程**(event dispatch thread)。

所有响应组件事件的监听器算法代码都会在事件分发线程中执行。例如,程序首先为界面上的图形组件注册响应事件的监听器;当用户操作图形组件时,Java 虚拟机会自动执行其注册监听器中的算法代码,对用户事件进行处理和响应。swing 框架将所有监听器算法代码都安排在事件分发线程中运行。图 8-20 给出一个图形用户界面程序中主线程和事件分发线程的运行示意图。

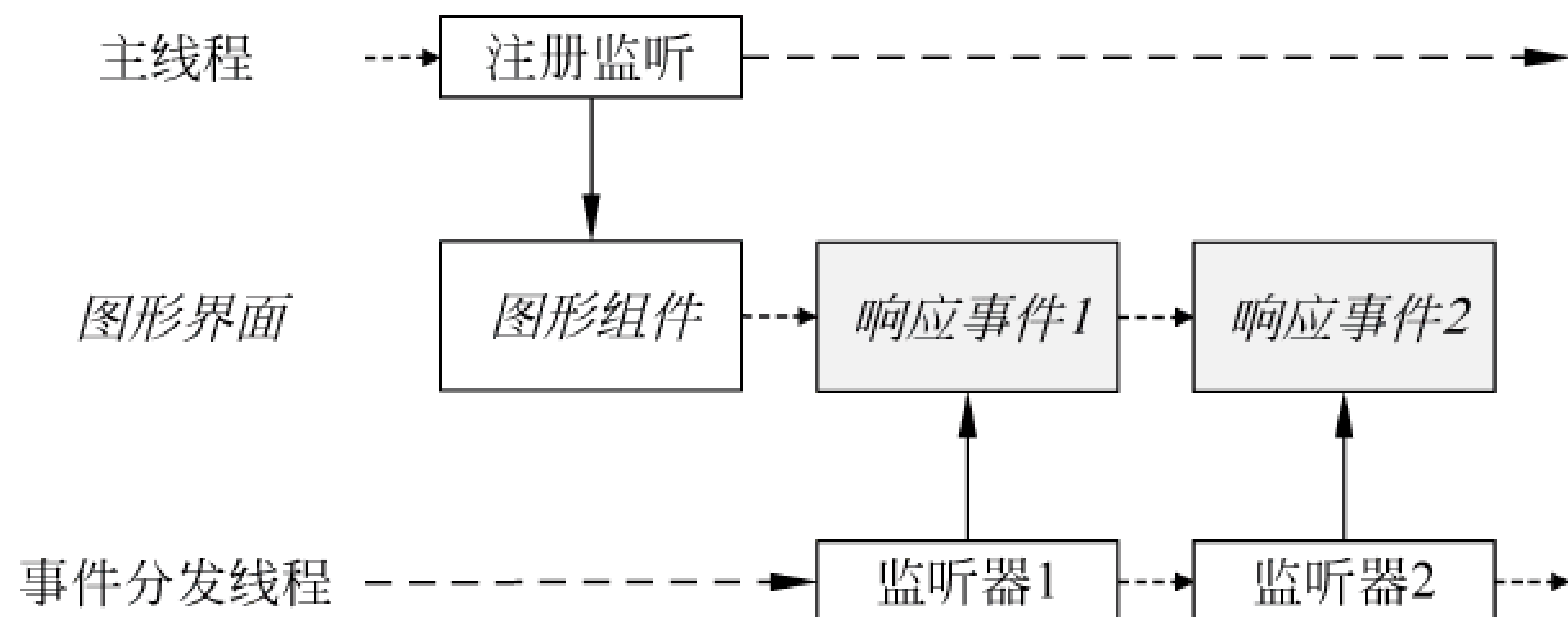


图 8-20　图形用户界面程序中主线程和事件分发线程的运行示意图

在事件分发线程中运行的监听器算法代码应能在短时间内完成执行，这样图形界面程序才会响应迅速，给用户提供良好的操作体验。所有其他费时的算法代码，例如加载较大的图像文件等，都应尽可能地放在其他线程中执行。

8.6.2　在线程中操作图形组件

程序员可以在图形用户界面程序中创建线程，然后在线程中操作界面上的图形组件，例如在图形组件上显示信息或绘图。图 8-21 给出一个在线程中操作图形组件的示例程序。

图 8-21 示例程序的主窗口包含一个 JButton 按钮和一个显示信息用的 JLabel 标签。示例程序在子线程中持续向 JLabel 标签显示一个倒计时信息“倒计时数：Hello from **Thread.**”。如果此时用户单击 JButton 按钮“Hello”，则 Java 虚拟机又会在事件分发线程中调用该按钮监听器的 actionPerformed()方法，向 JLabel 标签显示一个按钮信息“Hello from **Button.**”。换句话说，子线程和事件分发线程会并发向 JLabel 标签显示信息。例 8-14 给出了这个示例程序的完整 Java 演示代码。



图 8-21　在线程中操作图形组件的示例程序

例 8-14　在线程中操作图形组件的完整 Java 演示代码(JSwingInThread.java)

```
 1  import java.awt.*;                                  //导入 java.awt 包中的类
 2  import java.awt.event.*;                            //导入 java.awt.event 包中定义的事件类
 3  import javax.swing.*;                               //导入 javax.swing 包中的类
 4  public class JSwingInThread {                       //测试类
 5      public static void main(String[] args) {        //主方法
 6          JFrame w = new JFrame("Swing in threads");//创建程序主窗口
 7          w.setSize(400, 200);  w.setLocation(100,  100);w.setVisible(true);
 8          w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 9          //在窗口中添加一个 JButton 按钮,一个 JLabel 标签
10          JButton btn = new JButton("Hello");         //创建一个按钮
11          JLabel msg = new JLabel();                  //创建一个显示信息用的标签
12          w.add(btn, BorderLayout.NORTH );  w.add(msg, BorderLayout.CENTER );
13          w.validate();
14          //单击按钮时在标签上显示信息:为按钮添加响应 ActionEvent 事件的监听器
15          ActionListeneral = new ActionListener() { //匿名类:创建监听器对象
16              public void actionPerformed(ActionEvent e)  //实现接口规定的抽象方法
17              {  msg.setText("Hello from Button.");  }  //事件分发线程:在标签上显示信息
18          };
19          btn.addActionListener( al );                //为按钮 btn 添加监听器 bl
20          //下面创建子线程,运行倒计时算法对象,在标签上持续显示倒计时信息
21          CountDown ra = new CountDown(msg);          //创建一个倒计时算法对象 ra
22          Thread t = new Thread(ra);  t.start(); //创建并启动运行算法对象 ra 的子线程
23  }  }
```

```
24
25  class CountDown  implements Runnable {       //可在线程中运行的倒计时算法类
26      private JLabel msg;                      //显示倒计时信息的标签
27      private int count;                       //计数器
28      public CountDown(JLabel lab)             //构造方法
29      {  msg = lab;  }
30      public void run() {                      //实现 Runnable 接口规定的抽象方法
31          for (int n = 99; n >= 0; n-- ) {     //倒计时 100 次
32              count = n;
33              msg.setText(count + ": Hello from Thread.");
                                                 //子线程:在标签上显示倒计时信息
34              try {  Thread.sleep(500);  }     //休眠 0.5 秒
35              catch (InterruptedException e) { }
36          }
37  }  }
```

执行例 8-14 的示例程序,图 8-22 给出了其多线程运行示意图。

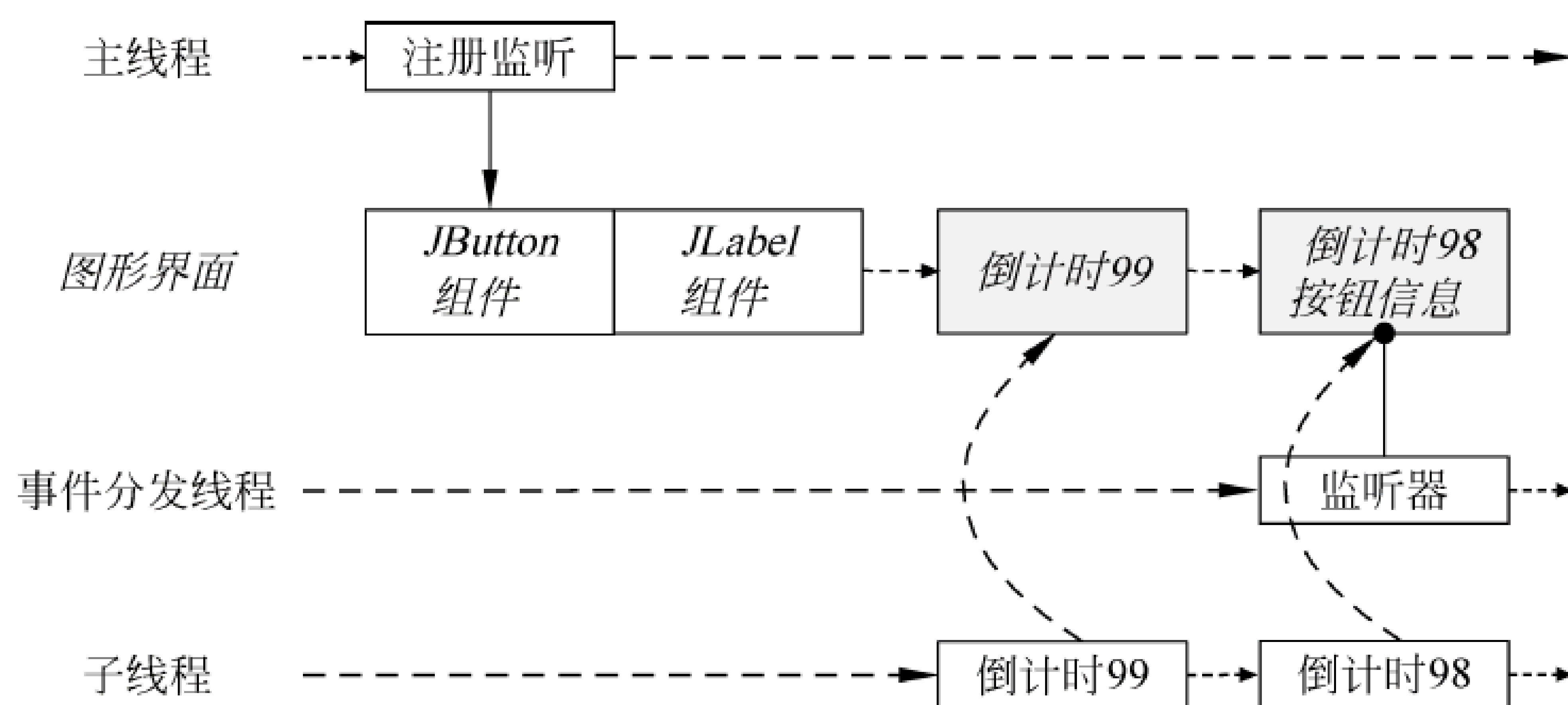


图 8-22 例 8-14 示例程序的多线程运行示意图

例 8-14 示例程序包含 3 个线程:主线程、事件分发线程和一个显示倒计时信息的子线程。从图 8-22 可以看出,子线程和事件分发线程并发操作 JLabel 标签组件,可能会出现同时向标签组件显示信息的情况。swing 框架中的图形组件类大多都不是线程安全的。在多个线程中并发操作同一个图形组件可能会出现互斥操作错误,而且这些错误是偶发的,难以重复,难以排查。为此,swing **框架要求**:在线程中操作图形组件,应当将算法代码交由事件分发线程统一执行。

8.6.3 通过事件分发线程操作图形组件

编写图形用户界面程序,如果需要在线程中操作图形组件,则应当将算法代码提交给事件分发线程,然后由事件分发线程统一调度、执行。具体的做法如下。

(1) 实现 **Runnable** 接口,将操作图形组件的算法代码包装成一个可在线程中运行的算法对象。

(2) 调用 javax.swing.**SwingUtilities** 类中的静态方法 **invokeLater**(),将算法对象提交给事件分发线程统一调度、执行。

所有响应组件事件的监听器算法代码,以及经由 invokeLater()方法提交的操作图形组

件算法代码都会在事件分发线程执行。当有多个算法代码需要执行时,事件分发线程会按照时间先后对它们进行排队,串行执行(如图 8-23 所示),这就避免了操作图形组件时的相互冲突。



图 8-23　事件分发线程中的运行调度

例 8-14 中代码第 33 行是在子线程中直接操作标签组件 msg,显示倒计时信息。

```
msg.setText(count + ": Hello from Thread.");        //子线程:在标签上显示倒计时信息
```

应当将这条操作标签组件 msg 的算法代码包装成一个可在线程中运行的算法对象,然后调用方法 invokeLater(),将其提交给事件分发线程执行。例如:

```
SwingUtilities.invokeLater( new Runnable() {        //使用匿名类形式包装算法对象,然后提交执行
    public void run() {                             //实现接口 Runnable 规定的抽象方法
        msg.setText(count + ": Hello from Thread."); //被包装的算法代码
    }
});
```

也可以使用匿名方法(Lambda 表达式)将上述算法代码包装成算法对象。例如:

```
SwingUtilities.invokeLater( () -> { msg.setText(count + ": Hello from Thread."); } );
```

注:关于匿名类和匿名方法的写法,请参见 4.6 节。

按上述方法修改例 8-14 中代码第 33 行,将操作标签组件 msg 的代码包装成算法对象并提交给事件分发线程执行。执行修改后的程序,图 8-24 给出了其多线程运行示意图。

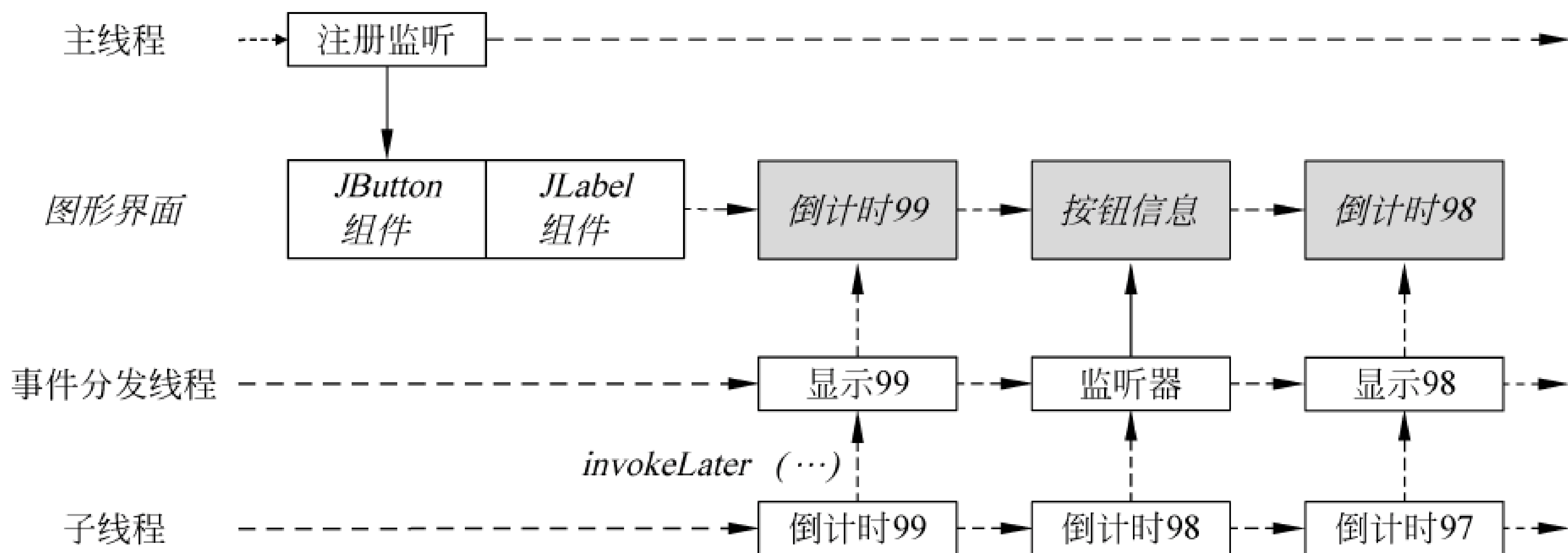


图 8-24　例 8-14 示例程序修改后的多线程运行示意图

从图 8-24 可以看出,在子线程中调用方法 invokeLater()就是将显示倒计时信息的算法代码交由事件分发线程执行。事件分发线程会统一调度监听器算法和倒计时算法,排队

执行。例如,当用户单击图形界面中的按钮 Hello 时,按钮信息和倒计时信息会交替显示。

8.6.4 多线程并发绘图

本节再给出一个多线程并发绘图的示例程序。多线程并发绘图时,需要将绘图算法包装成可在线程中运行的算法对象,然后调用方法 invokeLater()将其提交给事件分发线程去统一调度、执行。例 8-15 给出一个完整的多线程并发绘图程序 Java 演示代码。

例 8-15 一个完整的多线程并发绘图程序 Java 演示代码(JDrawInThread.java)

```
 1  import java.awt.*;                                          //导入 java.awt 包中的类
 2  import javax.swing.*;                                       //导入 javax.swing 包中的类
 3  public class JDrawInThread {                                //测试类:在两个线程中同时绘图
 4      public static void main(String [] args) {               //主方法
 5          JFrame w = new JFrame("在线程中绘图");               //创建程序主窗口
 6          w.setSize(600,400);  w.setLayout(null);  w.setVisible(true);
 7          w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 8          //创建绘制长方形和椭圆的绘图算法对象,然后放入两个子线程中运行
 9          DrawRunnable d1 = new DrawRunnable(w, 1);           //绘制长方形的算法对象 d1
10          DrawRunnable d2 = new DrawRunnable(w, 2);           //绘制椭圆的算法对象 d2
11          Thread t1 = new Thread(d1), t2 = new Thread(d2);
12          t1.start();  t2.start();
13  } }
14  class DrawRunnable implements Runnable {                    //可在线程中运行的绘图算法类
15      private JFrame win;                                     //在窗口 win 中绘图
16      private int x, y, w = 40, h = 30;                       //图形位置和大小
17      private int shape;                                      //图形:1 - 长方形,2 - 椭圆形
18      private boolean isStop = false;                         //超出窗口边界时停止绘图
19      public DrawRunnable(JFrame f, int s) {                  //构造方法
20          win = f;  shape = s;
21          if (shape == 1){x = 0;y = win.getHeight()/2;} //1 - 长方形:从左到右绘图
22          else           {x = win.getWidth()/2;y = 0;}//2 - 椭圆形:从上到下绘图
23      }
24      public void drawShape() {                               //在窗口 win 中绘图的方法
25          Graphics g = win.getGraphics();                     //获取窗口 win 的绘图对象
26          if (shape == 1) {                                   //绘制长方形
27              g.setColor( Color.WHITE );  g.fillRect(x, y, w, h);
28              g.setColor( Color.BLACK );  g.drawRect(x, y, w, h);
29          }
30          else {                                              //绘制椭圆形
31              g.setColor( Color.GRAY );   g.fillOval(x, y, w, h);
32              g.setColor( Color.BLACK );  g.drawOval(x, y, w, h);
33          }
34          if (shape == 1)  x += w;                            //长方形向右移动
35          else  y += h;                                       //椭圆形向下移动
36          if (x >= win.getWidth() || y >= win.getHeight() )  isStop = true;
                                                                //超出边界时停止绘图
37      }
38      public void run() {                     //实现抽象方法 run(),描述在线程中运行的绘图算法
```

```
39            while (isStop == false) {                    //循环绘图,直到超出窗口边界
40                SwingUtilities.invokeLater( () ->{   drawShape();   } );   //在线程中绘图
41                try { Thread.sleep(200); }   catch(InterruptedException e) { } //休眠 0.2 秒
42            }
43    }   }
```

在 Eclipse 集成开发环境中运行例 8-15 的程序,运行结果如图 8-25 所示。请读者对比运行结果去阅读理解例 8-15 中的程序代码。

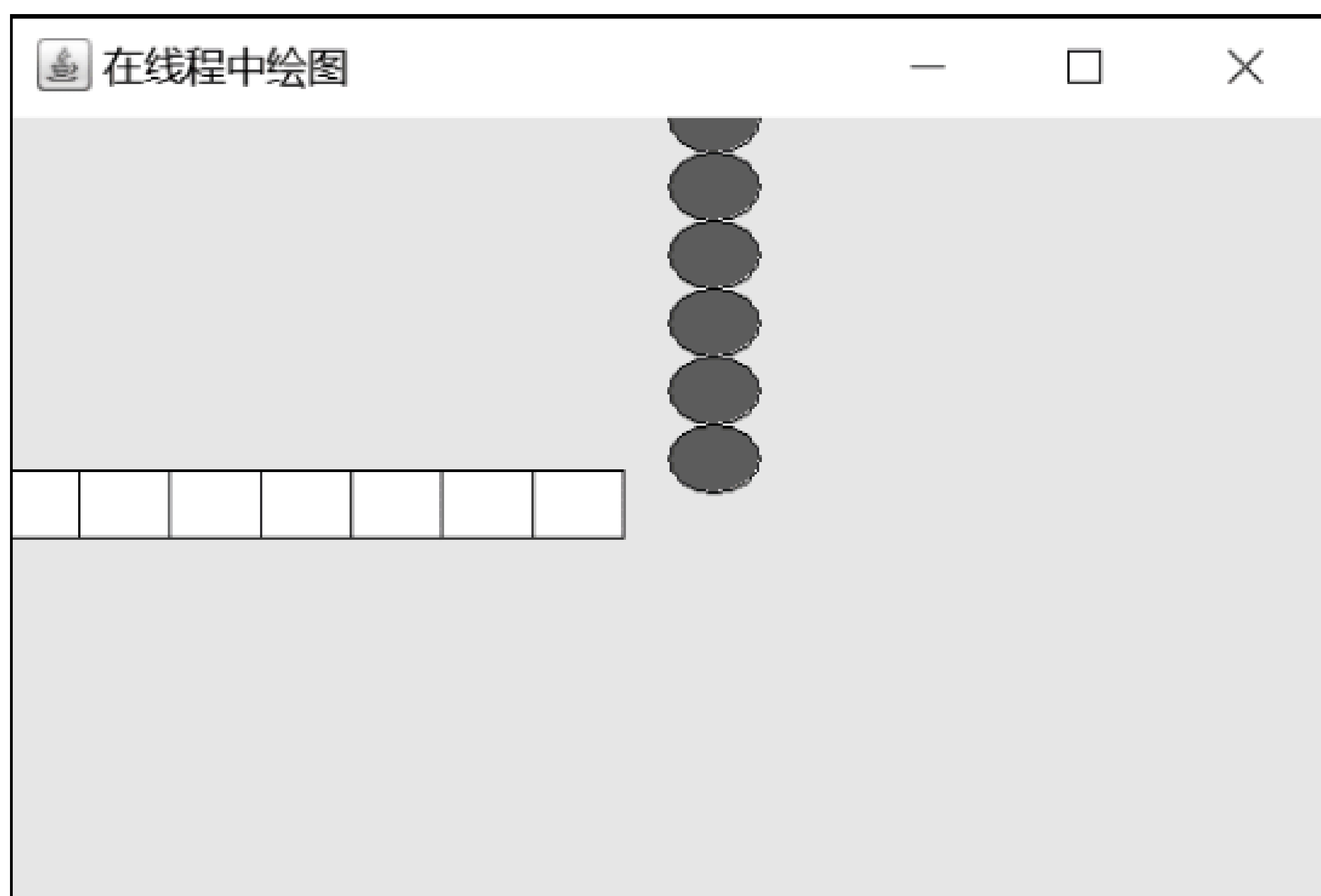


图 8-25　例 8-15 程序的运行结果

本节习题

1. 下列关于 swing 框架中事件分发线程的描述中,错误的是(　　)。
   A. 所有响应组件事件的监听器算法代码都会在事件分发线程中执行
   B. 在事件分发线程中运行的监听器算法代码应能在短时间内执行结束
   C. 在线程中操作图形组件应当将算法代码交由事件分发线程统一执行
   D. 应当将比较费时的算法代码放在事件分发线程中执行
2. 在使用 swing 框架编写的图形用户界面程序中,执行监听器算法代码的线程是(　　)。
   A. 主线程　　　　B. 事件分发线程
   C. 程序员创建的子线程　　　　D. 空线程
3. 使用 swing 框架编写的图形用户界面程序运行时至少包含(　　)个线程。
   A. 0　　　B. 1　　　C. 2　　　D. 4
4. 下列关于在线程中操作图形组件的描述中,错误的是(　　)。
   A. swing 框架中的图形组件类都是线程安全的类
   B. 应当将操作图形组件的算法代码包装成一个可在线程中运行的算法对象
   C. 应当将操作图形组件的算法代码提交给事件分发线程统一调度、执行
   D. 所有监听器以及操作图形组件的算法代码都应当在事件分发线程中执行
5. 将算法代码提交给事件分发线程统一调度执行的方法是(　　)。
   A. javax. swing. SwingUtilities 类中的静态方法 invokeLater()

B. java.awt.SwingUtilities 类中的静态方法 run()
C. javax.lang.Thread 类中的静态方法 invokeLater()
D. javax.lang.Thread 类中的静态方法 run()

本章学习要点

- 多线程是一种高级编程技术。多线程可以提高 CPU 使用率,改善用户体验。在多核或多 CPU 计算机系统上,使用多线程可以明显提高程序的运行速度。
- 要准确理解多线程编程中的三个要素。
  - ◇ 可以运行的**算法对象**,算法对象具有 run()方法。
  - ◇ 运行算法对象的**线程对象**,线程对象是 Thread 类的对象。
  - ◇ 被多个线程共享的**数据对象**,操作这些数据对象时需要启用同步(synchronized)机制,多线程协同还需要使用**等待-唤醒**(wait-notify)机制。
- 多线程编程比较复杂,学习时应仔细阅读并理解本章提供的示例程序,然后尝试自己重写一遍。

本章习题

1. 重写程序。阅读并重写 8.1.2 节例 8-1 的单线程串行程序和 8.1.3 节例 8-2 的多线程并发程序,通过比对运行结果来理解多线程与单线程之间的区别。

2. 重写程序。阅读并理解 8.3.2 节例 8-6 中的多线程并发售票服务程序,然后重写程序,记录程序的运行结果。修改程序,对其中售票窗口类 TicketWindow 的售票方法 sale()进行同步。通过比对修改前后的运行结果来理解 Java 语言的同步机制。

3. 重写程序。阅读并理解 8.4.3 节例 8-11 中的多线程"生产者-消费者"模式数据处理程序,然后重写这个程序。通过比对程序源代码与运行结果之间的关系来理解 Java 语言的"等待-唤醒"机制。

4. 重写程序。阅读并理解 8.5.2 节例 8-13 中的定时显示本地系统时间程序,然后重写这个程序,查看程序的运行结果。

5. 重写程序。阅读并理解 8.6.4 节例 8-15 中的多线程并发绘图程序,然后重写这个程序,查看程序的运行结果。

第9章 网络编程

当今世界，计算机网络无处不在，网络编程也成为程序员应当学习的一项非常重要的内容。本章学习网络编程。让我们先来了解一下计算机网络，图 9-1 给出了一个计算机网络的全景图。

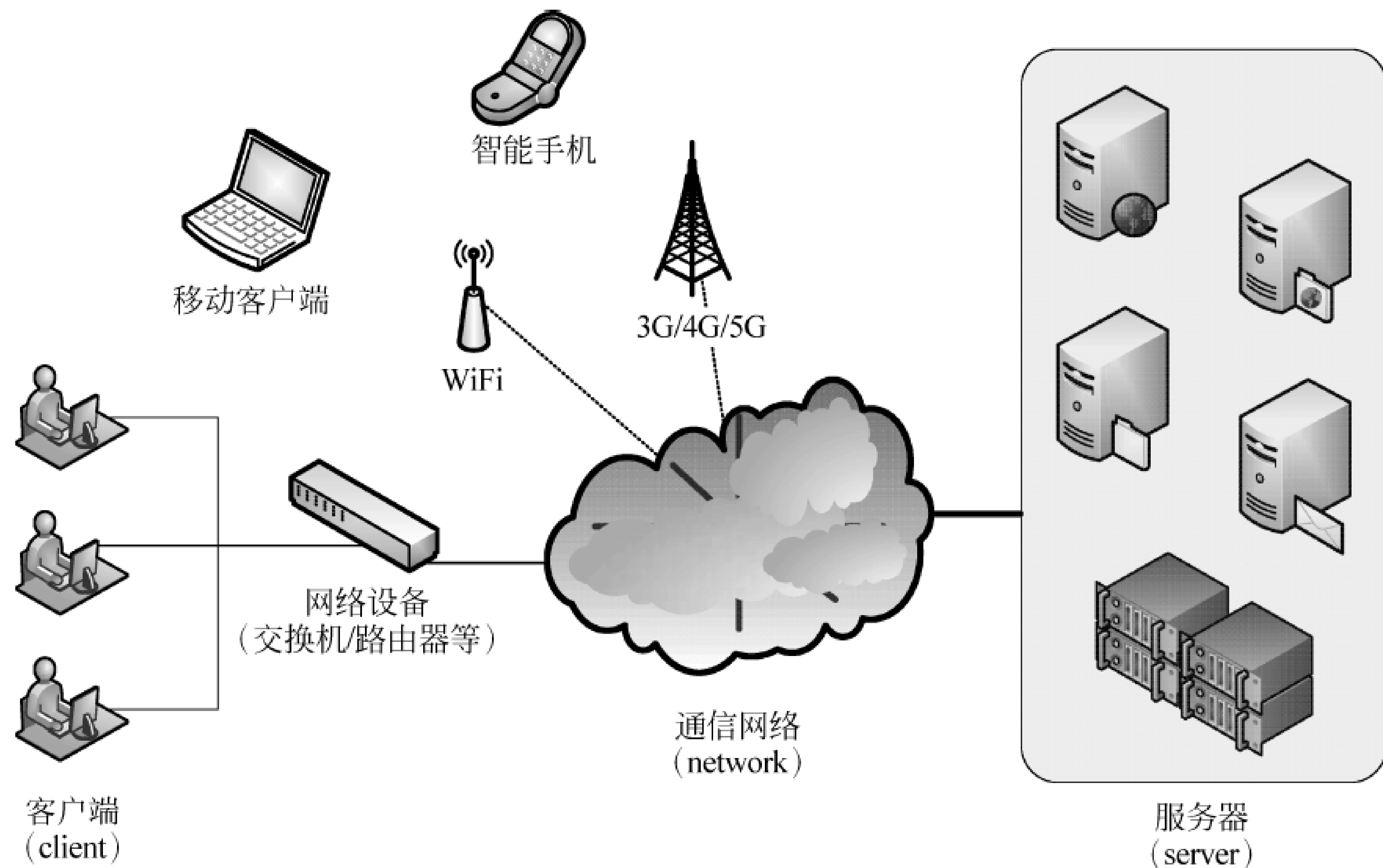


图 9-1　**计算机网络全景图**

计算机网络由三大要素组成，它们分别是**通信网络**、**服务器**和**客户端**。

（1）通信网络。

遍布全球的通信线路、无线基站、跨洋光缆、通信卫星以及各种各样的网络设备，它们共同构成了一个可以互联互通的**通信网络**（network）。

（2）服务器。

网络上有一些专门提供**服务**（service）的计算机。因为它们提供服务，所以被称为**服务器**（server）。服务器在本质上就是一台计算机，是一个由硬件和软件组成的计算机系统。

因为要同时向很多用户提供服务，服务器**硬件**通常需要有更强的计算能力和存储能力，例如装配多个 CPU、具有更大的内存和更多的硬盘。服务器还需要使用性能更强、安全性更高的**操作系统**，例如 Linux、UNIX 或 Windows Server 等。

服务器与普通计算机的最大区别在于**应用程序**。网络服务有很多种，例如 **WWW**

(World Wide Web,**Web**)服务(即网站服务)、**E-mail** 电子邮件服务(即收发电子邮件)、**FTP**(File Transfer Protocol)文件传输服务(即文件上传与下载)等。网络服务是通过**服务器应用程序**来提供的,不同网络服务需要不同的服务器应用程序。例如,提供 WWW 网站服务需要 Web 服务器程序(如 IIS for Windows Server),提供 E-mail 电子邮件服务需要 Mail 服务器程序,提供 FTP 文件传输服务需要 FTP 服务器程序等。

(3) 客户端。

使用网络服务的计算机系统被称为**客户端**(client)。客户端可以是台式计算机、笔记本电脑或智能手机等。使用网络服务是通过**客户端应用程序**实现的,不同网络服务可能需要使用不同的客户端应用程序。例如,使用 WWW 网站服务需要 Web 客户端程序(例如 IE 浏览器),使用 E-mail 电子邮件服务需要 Mail 客户端程序(例如 Outlook),使用 FTP 文件传输服务需要 FTP 客户端程序等。计算机上最常用的客户端应用程序是**浏览器**(browser),而智能手机上则是各种各样的 **App**。App 是 Application(应用)的昵称,它实际上就是网络服务中的客户端应用程序。

一个完整的网络服务由客户端应用程序和服务器应用程序两部分组成,称为 **Client/Server** 程序架构,简称 **C/S 架构**。学习网络编程,首先需要了解计算机网络的基本原理,然后学习如何编写**网络应用程序**,其中包括客户端应用程序和服务器应用程序。

9.1 计算机网络的基本原理

计算机网络是计算机专业一门独立的课程,课程内容很多,也很专业。很多读者在学习程序设计之前并没有学过计算机网络课程,不具备学习网络编程的基础。

针对上述问题,本节抽丝剥茧,将程序员必须具备的网络知识提炼出来,以通俗易懂的形式呈现给读者。在掌握了这些网络知识之后,读者就可以无障碍地学习本章后续网络编程部分的内容了。

9.1.1 TCP/IP

浏览器是一个常用的客户端应用程序。在浏览器中输入中国农业大学网站服务器的网址“http://www.cau.edu.cn”,就可以看到中国农业大学网站的主页,如图 9-2 所示。

浏览器是如何获得中国农业大学 WWW 网站服务器上的网页信息的呢?浏览器是一个客户端应用程序。浏览器获取服务器信息的过程实际上是一个**客户端应用程序**与**服务器应用程序**互相通信的过程。这个通信过程通常由客户端应用程序发起,客户端应用程序(例如 IE 浏览器)首先向网址所指定的服务器应用程序发送服务请求;服务器应用程序(例如 IIS Server)接收请求,然后将所请求的信息(例如网站主页)发送回客户端应用程序。

程序间相互通信需要通过**计算机网络**来完成。计算机网络是一个非常复杂的系统。大到通信原理、操作系统,小到网络地址格式、网线插头(俗称水晶头)的外观及尺寸等,所涉及的内容非常多。如何让这样一个复杂系统中的各种软硬件产品有机地结合到一起,协同工作呢?为此,国际互联网协会(Internet Society,ISOC)资助制定了一系列互联网规范和标准,术语称为**协议**(protocol)。这些协议以 RFC(Request For Comments)文件的形式在因



图 9-2 中国农业大学网站的主页

特网上公开发布。

互联网协议是一系列协议的集合。其中有两个非常重要的基础协议,分别是**传输控制协议**(Transmission Control Protocol,**TCP**)和**网间互连协议**(Internet Protocol,**IP**)。因此在互联网行业,互联网协议被称作 **TCP/IP**。今天我们所使用的**因特网**(Internet)就是基于 TCP/IP 建立起来的国际互联网,所有在因特网中使用的软硬件产品必须遵守 TCP/IP。

可以按功能将 TCP/IP 划分成四层,称为 TCP/IP 网络四层模型,如图 9-3 所示。最高层是用户直接使用并能感知到的**应用层**,然后是**传输层**、**网络层**,最底层是最终实现通信功能的**链路层**。

下面结合图 9-3 来具体讲解 TCP/IP 网络模型中各层的功能及其主要协议。

9.1.2 应用层

一个完整的网络服务由客户端应用程序和服务器应用程序两部分组成。这两个应用程序必须遵守共同的规范或标准,这样才能协同工作,实现网络服务的功能。

TCP/IP 网络的**应用层**(application layer)协议就是一组关于网络服务的规范和标准,其中包括 Web 服务、E-mail 电子邮件服务、FTP 文件传输服务等常用的网络服务协议。应用层协议用于规范网络服务的内容及服务流程。

1. 与 Web 服务相关的应用层协议

1) HTTP

HTTP(HyperText Transport Protocol)是**超文本传输协议**的简称,其中规范了浏览器

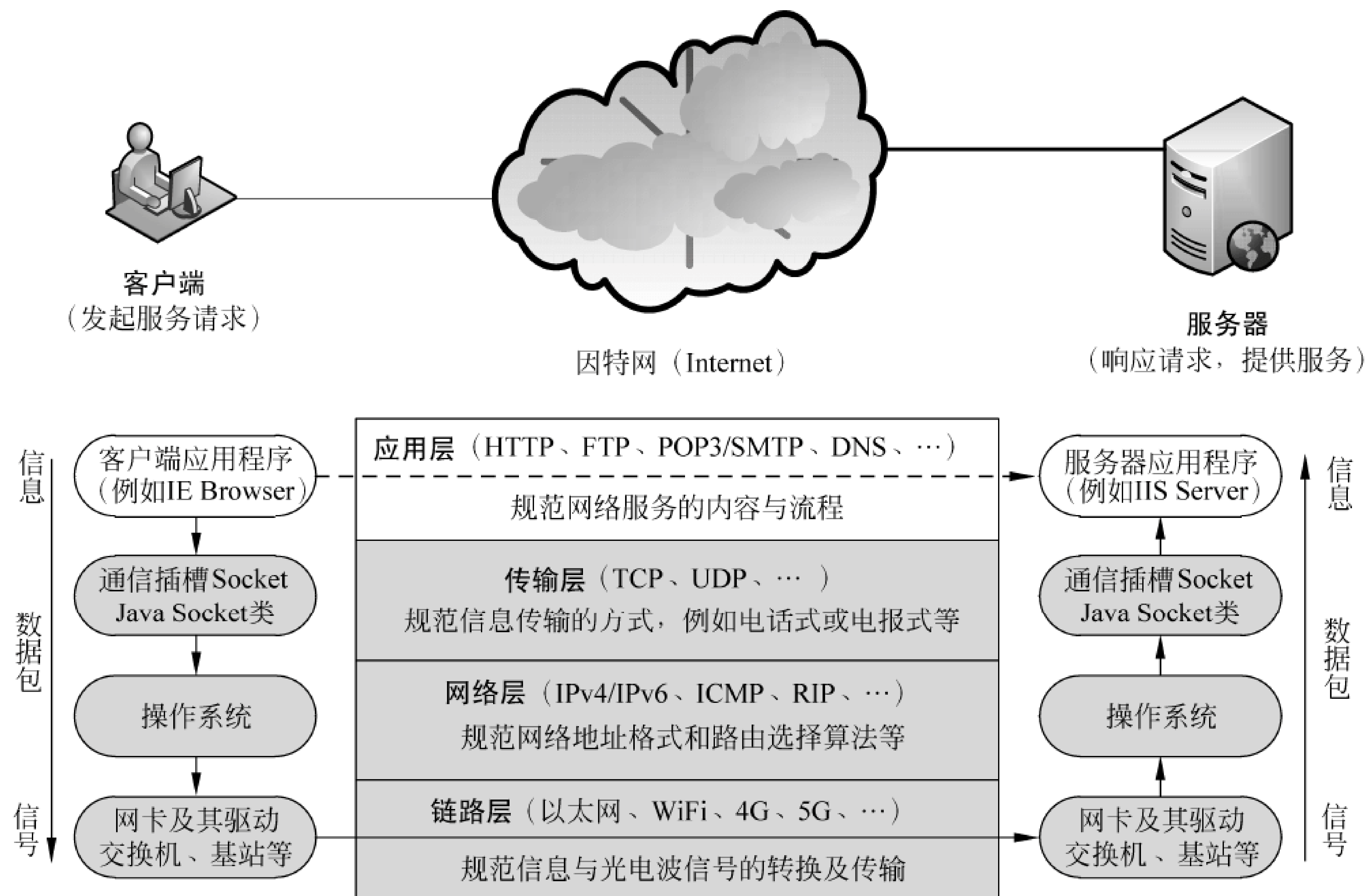


图 9-3 TCP/IP 网络四层模型

和 Web 服务器之间的服务请求-响应流程。例如访问网站时，浏览器会先向服务器发送一个 HTTP 请求。HTTP 请求是一串文本信息，其原文看起来类似下面的样子(具体含义请查阅 HTTP)。

```
GET /index.html HTTP/1.1
Accept: */*
Accept-Language: zh-CN
Host: localhost:8000
Connection: Keep-Alive
```

Web 服务器在收到上述请求后会回复一个 HTTP 响应，并将所请求的网页文件(例如 index.html)发送回浏览器。浏览器接收到的 HTTP 响应及网页文件(HTML 格式)看起来类似下面的样子(具体含义请查阅 HTTP)。

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
<html>
    <head><title>Web 服务 HTML 测试页面</title></head>
    <body>
        <p>Hello World!</p>
        <p>你好,世界!</p>
    </body>
</html>
```

这串文本信息是浏览器接收到的原文。浏览器会对这段原文进行解析，提取并显示出其中的网页内容，如图 9-4 所示。

2）HTML

HTML（HyperText Markup Language）是超文本标记语言的简称，其中规范了 HTML 网页的内容及语法格式。如需了解 HTML，请查阅相关的书籍或资料。

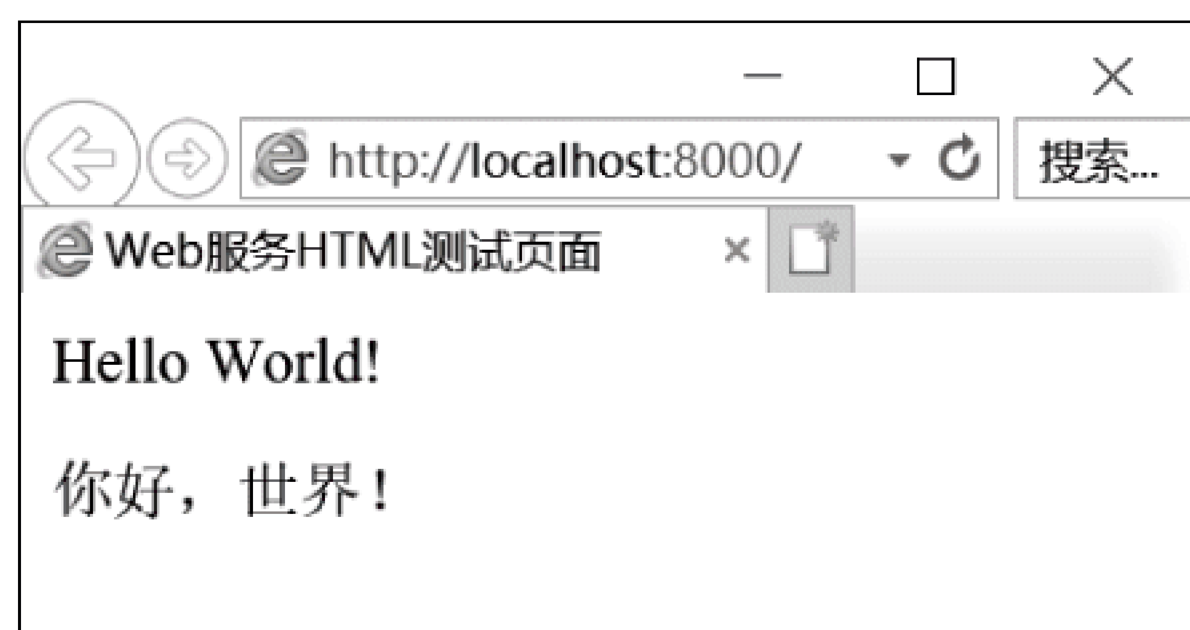


图 9-4 浏览器对 HTTP 响应原文进行解析并显示出其中的网页内容

3）DNS 域名系统

网络上的一台计算机称为一个**主机**(host)，每个主机都有一个**主机名**(host name)。为了提供网络服务，因特网上对外提供服务的服务器通常都需要有一个便于记忆并经权威机构认证登记的名字，这个名字称作服务器的**域名**(domain name)。服务器的域名就是其对外的主机名。以下是一些因特网服务器域名的例子。

www.baidu.com，百度搜索引擎服务器的域名。

www.oracle.com，甲骨文公司(Java 所有权人)网站服务器的域名。

www.cau.edu.cn，中国农业大学网站服务器的域名。

www.icourse163.org，中国大学 MOOC 在线教育平台服务器的域名。

localhost，这是一个特殊的域名，用于表示用户当前使用的计算机(称作"本机")。

域名便于人的记忆，但计算机网络内部使用的则是数值形式的网络地址(称作 IP 地址，参见 9.1.4 节)。**DNS**(Domain Name System)域名系统是一个关于域名的应用层协议，其中规范了域名的格式、域名与 IP 地址之间如何映射等。

2. 其他应用层协议

除了与 Web 服务相关的协议之外，TCP/IP 网络应用层还包括很多其他网络服务协议，常用的如下。

- **FTP**(File Transfer Protocol，文件传输协议)，其中规范了文件下载、上传和账户控制等服务的请求-响应流程。
- **SMTP**(Simple Mail Transfer Protocol，简单邮件传输协议)，其中规范了电子邮件的发送和传递服务流程。
- **POP3**(Post Office Protocol 3，邮局协议第 3 版)，其中规范了电子邮件的查询和接收服务流程。

3. 程序员与应用层协议的关系

如果需要编写 Web 服务、E-mail 电子邮件服务或 FTP 文件传输服务等通用网络服务程序，程序员首先应当了解 TCP/IP 网络中相关的应用层协议，然后按照协议要求编写程序。例如，如果想自己编写一个新的浏览器程序，那么程序员应当学习 HTTP 和 HTML 语法，然后按照要求设计、编写浏览器程序，只有这样才能正常访问互联网上的 Web 服务器。

如果想编写自己专有的网络服务程序，例如设计一种新的聊天服务程序，程序员可以制定自己的应用层协议，明确聊天服务的内容和流程，然后按照要求分别编写客户端应用程序和服务器应用程序。这两个应用程序遵守同一种协议，它们之间可以互相通信，共同实现新

的网络聊天服务。

9.1.3 传输层

应用层协议规范了网络服务的内容及服务流程。例如,针对 Web 服务,HTML 规范了超文本文件的内容及语法格式,HTTP 则规范了浏览器和 Web 服务器之间的服务请求-响应流程。但是应用层协议并没有涉及客户端应用程序与服务器应用程序之间具体的通信过程,它们之间是如何进行通信的呢?例如,浏览器如何将 HTTP 请求信息发送给 Web 服务器,Web 服务器又是如何将 HTTP 响应信息发送回浏览器的呢?

TCP/IP 网络的**传输层**(transport layer)协议就是一组关于客户端应用程序和服务器应用程序之间互相通信的规范和标准。TCP/IP 网络提供了两种不同的通信方式,分别是**有连接**通信 TCP 和**无连接**通信 UDP。

1. TCP 和 UDP

TCP/IP 网络提供的第一种通信方式是有连接通信,即通信双方先建立**连接**(connection),然后再进行双向数据传输。这有点类似于打电话,打电话之前需先拨通电话(即建立连接),然后两个人之间进行双向通话。**传输控制协议**(Transmission Control Protocol,**TCP**)就是传输层中关于有连接通信的规范或标准。

TCP/IP 网络提供的第二种通信方式是无连接通信,它直接将数据单向传输给对方,不需要事先建立连接,事后也不需要对方回复。这有点类似于发电报,在电报上标明收报人并通过电波单向发送出去;收报人接收电报,本次发报过程就结束了。如果在电报上标明的不是某个特定的收报人,而是一个群组,则所有加入该群组的收报人都可以接收电报,这就变成了**多播**(multicast,或称**组播**)。**用户数据报协议**(User Datagram Protocol,**UDP**)就是传输层中关于无连接通信的规范或标准。

TCP 有连接通信为应用层提供了一种可靠的数据传输方式。例如,HTTP 就需要使用 TCP 这种可靠的数据传输方式,这样访问网站时才不会丢失数据。而 UDP 无连接通信所提供的则不是百分之百可靠的数据传输。例如,网络视频直播使用的就是 UDP 组播,其优点是占用带宽少,但网络繁忙时有可能丢失数据,视频播放会出现卡顿现象。

TCP、UDP 提供的是一种被称作“点到点”或“端到端”的传输形式,即每次传输数据时都会有一个**发送方**(sender)和一个**接收方**(receiver)。网络通信的本质是网络应用程序之间的数据传输。当两个程序通过传输层进行网络通信时,其中一个程序是发送方,另一个程序则是接收方。

2. 传输层与应用层的关系

应用层协议用于规范网络服务的内容及服务流程。在应用层协议中,所有通信内容都有明确的含义,因此将应用层协议的通信内容称为**信息**。例如,HTTP 用于规范 Web 服务的内容及服务流程,其中明确了 HTTP 请求的内容及含义、HTTP 响应的内容及含义,它们分别被称为 HTTP 请求信息和 HTTP 响应信息。应用层信息需要通过传输层从发送方传输给接收方。

传输层协议用于规范发送方与接收方之间的通信方式。不管应用层传输什么信息,在传输层看来它们都是**数据**。传输层不关心这些信息是什么,它只负责传输。传输层会对需

要传输的应用层信息进行分拆、封装，将它们封装成一个个小的**数据包**(packet)，然后附加上一些控制信息再进行传输。

3. 端口

假设有两个网络应用程序 A、B，它们运行在同一台主机 Host1 上，如图 9-5 所示。

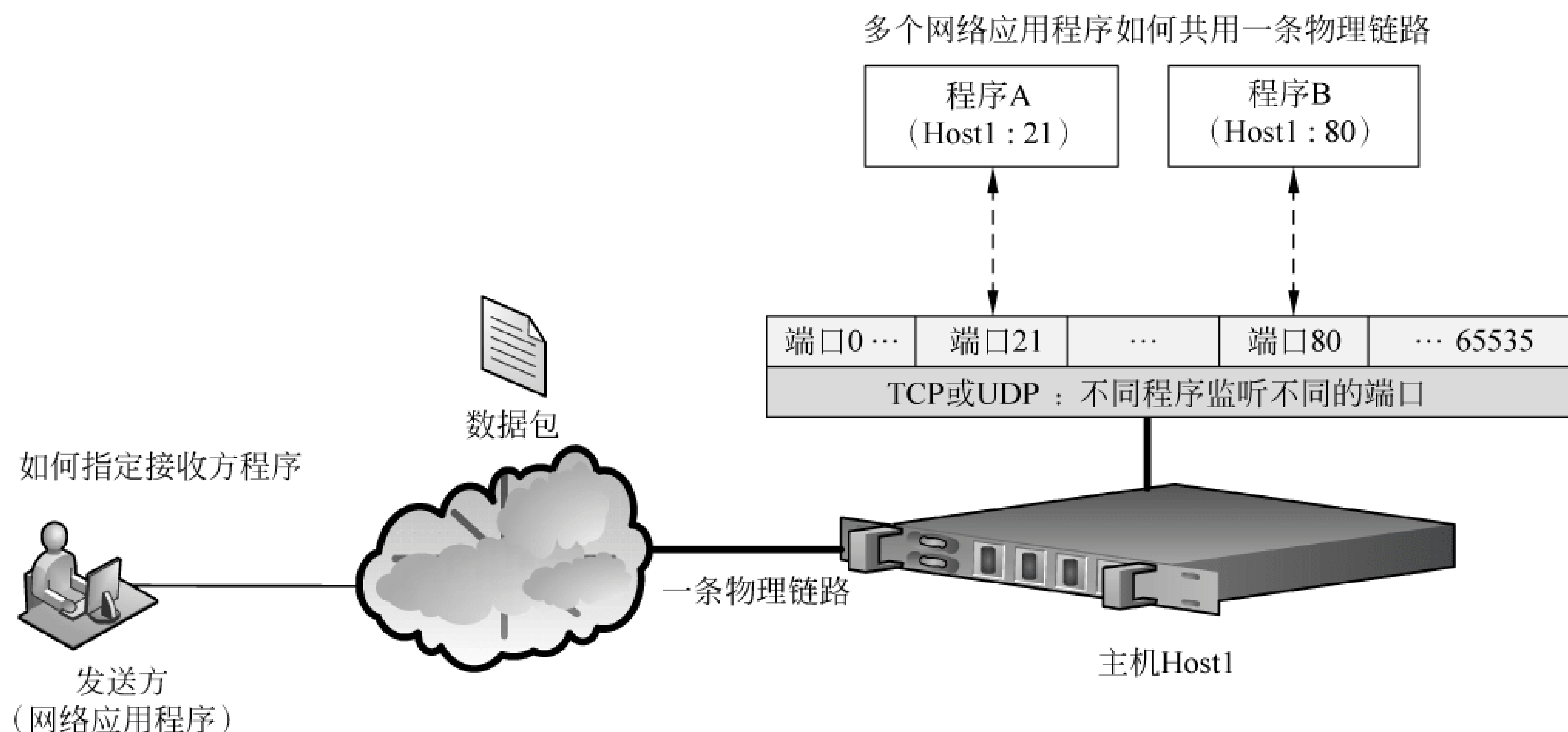


图 9-5 一台主机运行多个网络应用程序

针对图 9-5 的网络场景，有如下两个问题需要讨论。

(1) 多个网络应用程序如何共用一条物理链路?

通常，一台主机只有一条接入网络的物理链路。如果在一台主机上同时运行多个网络应用程序，它们都需要与外界进行网络通信。多个程序如何共用一条物理链路呢?

传输层基于同一物理链路划分出多个**端口**(port)，用于区分不同程序之间的通信。端口使用 16 位整数编号。同一物理链路可划分出 65536 个端口，端口号依次为 0～65535。

网络应用程序使用某个端口，检查并接收其通信数据，这被称作是**监听**(listen)端口。不同程序监听不同的端口，传输层根据端口号将所接收到的网络数据分发给对应的程序，这就实现了多个程序共用同一条物理链路。

(2) 发送方如何指定接收方程序?

如果需要向主机 Host1 上的程序 A 或程序 B 发送数据，发送方该如何区分这两个程序，或者说发送方该如何指定接收方程序呢?

网络上的每台主机都有自己的主机名(域名)和 IP 地址。发送方以“**主机名：端口号**”或“**IP 地址：端口号**”的形式来指定接收方程序。其中主机名或 IP 地址用于指定接收方的主机，端口号用于指定该主机上对应端口的程序。

在图 9-5 中，假设主机 Host1 的主机名是“www.cau.edu.cn”，程序 A 监听端口 21，则程序 A 可以用“www.cau.edu.cn ：**21**”表示；程序 B 监听端口 80，则程序 B 可以用“www.cau.edu.cn ：**80**”表示。

4. 程序员与应用层、传输层的关系

应用层协议用于规范网络服务的内容及服务流程，而传输层协议用于规范发送方与接

收方之间的通信方式。

编写网络应用程序,程序员需按照应用层协议拟定通信内容,再根据传输层协议选择通信方式(例如 TCP 或 UDP),将通信内容从发送方传输到接收方。接收方程序也可能需要对接收到的通信内容进行回复。编写接收方程序的程序员也需要按照应用层协议的规定拟定回复内容,再通过传输层将回复内容传输给发送方。

9.1.4 网络层与链路层

TCP/IP 网络的**网络层**(network layer)协议主要用于规范网络间的寻址、路由选择、流量控制和拥塞处理,其中包括 IP、ICMP 等。

TCP/IP 网络的**链路层**(link layer)是网络通信最终的执行层,主要用于规范二进制数据流与光、电、波等模拟信号的转换及传输。TCP/IP 链路层本身并没有制定具体的协议,它主要依赖其他现成的基础通信网络标准,例如以太网、WiFi、4G、5G 等协议标准。

作为软件开发工程师,编写网络应用程序主要涉及应用层和传输层协议,通常不会直接用到网络层或链路层协议。如果想成为网络工程师,那就需要深入学习 TCP/IP 网络四层模型中的全部内容,这超出了本书的范围。本书面向软件开发工程师,即面向编写网络应用程序的程序员。

在本节,程序员唯一需要了解的内容是网络层 IP 中的网络地址,简称 IP 地址(IP address)。目前 IP 有两个版本,分别是 **IPv4** 和 **IPv6**。

- **IPv4 地址**。这是目前正在使用的网络地址,由 4 个 0～255 的十进制整数组成(4 个单字节整数,共 32 位),中间用点"."分隔。例如 **192.168.1.175**。
- **IPv6 地址**。这是一种新的网络地址格式,未来将用于解决 IPv4 地址不足的问题。IPv6 地址由 8 个 0～65535 的十六进制整数组成(8 个双字节整数,共 128 位),中间用冒号":"隔开。例如 **ABCD:EF01:2345:6789:ABCD:EF01:2345:6789**。

需要为接入计算机网络的每一台主机(包括计算机、智能手机等)**静态**或**动态**分配一个唯一的 IP 地址。计算机网络内部是使用 IP 地址来区分不同主机的。

除了 IP 地址之外,因特网上对外提供服务的服务器通常还需要有一个便于记忆的域名,即对外公开的主机名。域名也必须是唯一的,它必须经过权威机构的认证登记,然后才能生效。为了将域名转换成计算机网络内部使用的 IP 地址,因特网设置了一些专门的域名服务器(Domain Name Server,**DNS**)。

为了处理 IP 地址、主机名或域名,Java API 提供了一个因特网地址类 **InetAddress**。请读者阅读下面的因特网地址类 InetAddress 说明文档。

java.net.**InetAddress** 类说明文档

public class **InetAddress**
extends **Object**
implements **Serializable**

	修饰符	类成员(节选)	功能说明
1	static	InetAddress **getLocalHost**()	获取本机的因特网地址对象
2	static	InetAddress **getByName**(String host)	通过主机名创建因特网地址对象
3	static	InetAddress **getByAddress**(byte[] addr)	通过 IP 地址创建因特网地址对象

续表

	修 饰 符	类成员(节选)	功 能 说 明
4		byte[] **getAddress**()	获取 IP 地址
5		String **getHostAddress**()	获取字符串形式的 IP 地址
6		String **getHostName**()	获取主机名
7		boolean **isReachable**(int timeout)	检查因特网地址是否可以连通
		…	

例 9-1 给出了一个因特网地址类 InetAddress 的 Java 演示程序。

例 9-1 一个因特网地址类 InetAddress 的 Java 演示程序(JInetAddressTest.java)

```
1  import java.net.*;                          //导入 java.net 网络包中的类
2  public class JInetAddressTest {             //测试类:测试因特网地址类 InetAddress 的用法
3      public static void main(String[] args) {//主方法
4          try {                               //处理可能出现的勾选异常 UnknownHostException
5              InetAddress local = InetAddress.getLocalHost();
                                               //获取本机的因特网地址对象
6              System.out.println("通过 getLocalHost()获得本机因特网地址对象: " + local );
7              System.out.println("getHostName(): " + local.getHostName() );
                                               //主机名
8              System.out.println("getHostAddress(): " + local.getHostAddress() );
                                               //IP 地址
9              System.out.println();
10             //下面演示域名与主机名、IP 地址之间的关系
11             String cauWeb = "www.cau.edu.cn";              //中国农业大学网站的主机名
12             InetAddress cau = InetAddress.getByName(cauWeb); //根据主机名创建对象
13             System.out.println("根据主机名创建因特网地址对象: " + cau );
14             System.out.println("getHostName(): " + cau.getHostName() );  //主机名
15             System.out.println("getHostAddress(): " + cau.getHostAddress() );  //IP 地址
16         }
17         catch(UnknownHostException e) { e.printStackTrace(); } //捕捉并处理勾选异常
18 } }
```

在作者的计算机上运行例 9-1 的程序,运行结果如图 9-6 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> JInetAddressTest [Java Application] C:\Java\jre1.8.0
通过getLocalHost()获得本机因特网地址对象: Kan/192.168.1.7
getHostName(): Kan
getHostAddress(): 192.168.1.7

根据主机名创建因特网地址对象: www.cau.edu.cn/114.251.217.179
getHostName(): www.cau.edu.cn
getHostAddress(): 114.251.217.179
```

图 9-6 例 9-1 程序的运行结果

本节最后对 TCP/IP 做一个总结。

- TCP/IP 是一组关于网络服务及网络通信的规范和标准。

- TCP/IP 网络模型将 TCP/IP 按照功能划分成四层,分别是应用层、传输层、网络层和链路层。
- 编写网络应用程序主要涉及应用层和传输层协议,通常不会直接用到网络层或链路层协议。程序员应当了解应用层和传输层协议的主要内容和基本工作原理,特别是传输层的 TCP 和 UDP。
- Java API 提供了一组与网络编程相关的类,例如网络层的因特网地址类、传输层的套接字类、应用层的统一资源定位符类等。

在学习完计算机网络的基本原理之后,本章将进入网络编程环节的学习,正式学习如何利用 Java API 编写网络应用程序。

本节习题

1. 下列选项中,(　　)不属于计算机网络的范畴。
   A. 通信网络　B. 服务器　C. 客户端　D. 图形用户界面
2. 肯定能被网络应用程序用户感知到的 TCP/IP 层是(　　)。
   A. 应用层　B. 传输层　C. 网络层　D. 链路层
3. 编写网络应用程序通常不会涉及的 TCP/IP 层是(　　)。
   A. 应用层　B. 传输层　C. 网络层　D. 链路层
4. (　　)不属于 TCP/IP 网络的应用层协议。
   A. HTTP　B. FTP　C. POP3　D. IP
5. (　　)属于 TCP/IP 网络的传输层协议。
   A. HTTP　B. TCP　C. POP3　D. IP
6. (　　)属于 TCP/IP 网络的网络层协议。
   A. HTTP　B. TCP　C. POP3　D. IP
7. 下列选项中,(　　)不能被用于区分计算机网络上的不同主机。
   A. 主机名　B. 域名　C. IP 地址　D. 网络应用程序名
8. 下列选项中,被 TCP/IP 用于区分同一主机上不同网络应用程序的是(　　)。
   A. 主机名　B. IP 地址　C. 端口　D. 程序文件名

9.2 网络服务与网络资源

本节首先讲解网络服务及其所提供的网络资源,然后再讲解如何编写客户端应用程序访问网络资源。

9.2.1 网络服务

网络服务有很多种,常用的有 Web 服务、FTP 文件传输服务、E-mail 电子邮件服务、JDBC 数据库服务、Web Service 服务等。可以在一台服务器主机上安装多个服务器应用程序,这样就能用一台主机同时提供多个网络服务。

1. 网络服务的端口

为了区分同一主机上不同网络服务之间的通信，各服务器应用程序应当选择不同的传输层端口，这样它们就能通过一条物理链路同时向外界提供服务。简单地说，服务器可以通过不同端口向外界提供多个网络服务，如图 9-7 所示。

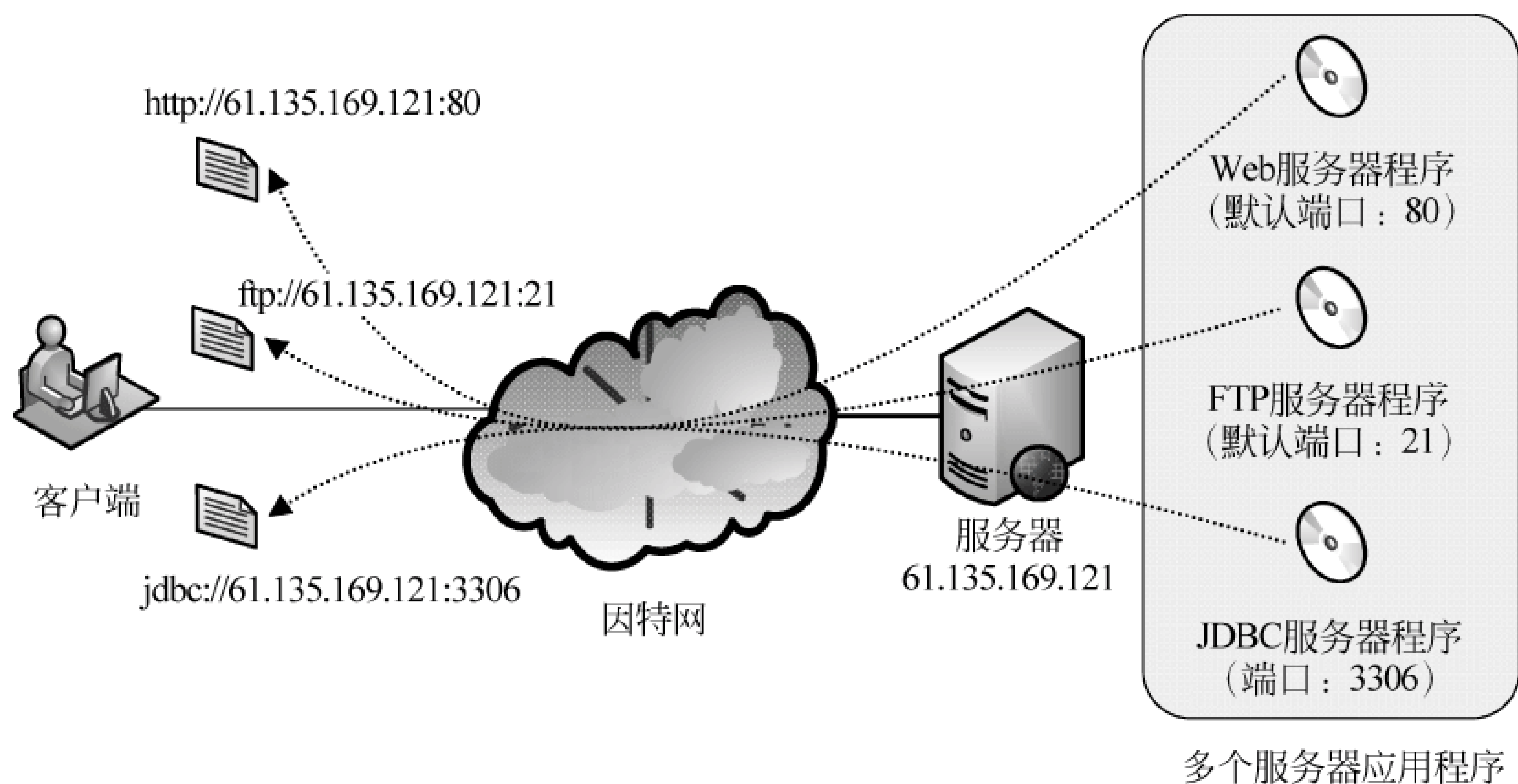


图 9-7　服务器通过不同端口向外界提供多个网络服务

一条物理链路可划分出 65536 个端口，端口号依次为 0～65535。理论上，服务器应用程序可以选用任意端口向外界提供服务。

为便于管理、减少冲突，互联网名称与数字分配组织（Internet Corporation for Assigned Names and Numbers，**ICANN**）将 0～1023 的 TCP 端口分配给了一些常用的网络服务。例如，TCP 80 端口被分配给了 Web 服务（HTTP），TCP 21 和 20 端口被分配给了 FTP 文件传输服务、TCP 25 和 110 端口被分配给了 E-mail 电子邮件服务。这些端口被称为**公认端口**（well known ports）。程序员在为自己的服务器应用程序选择端口时应避开公认端口，选择 1024～49151 的某个端口。

2. 网络服务的地址

可以用主机名或 IP 地址来表示网络上的一台主机。而要表示运行于主机上的一个网络服务时，则需要同时指明其主机、协议和端口。这三项内容合在一起就构成了一个网络服务的地址，其语法格式如下。

```
protocol: //host [: port]
```

其中，**protocol** 指明网络服务所使用的应用层协议，例如 http、https、ftp、file、jdbc 等；**host** 指明网络服务所在主机的主机名或 IP 地址；**port** 指明网络服务所使用的端口，缺省时使用默认的公认端口。网络服务是由服务器应用程序提供的。一个网络服务对应一个服务器应用程序，网络服务的地址实际上也就是提供该服务的服务器应用程序地址。

例如，图 9-7 中服务器主机的 IP 地址为“61.135.169.121”。该主机上同时运行了 3 个服务器应用程序，它们所提供的网络服务的地址分别如下。

- Web 服务的地址是“**http**://61.135.169.121:**80**”。
- FTP 服务的地址是“**ftp**://61.135.169.121:**21**”。
- JDBC 服务的地址是“**jdbc**://61.135.169.121:**3306**”。

客户端应用程序通过网络服务地址来指定希望使用的网络服务。

9.2.2 统一资源定位符

使用网络服务的目的是访问该服务所提供的**资源**(resource)。这些资源可能是服务器上的一个**数据文件**(例如 HTML 网页文件),也可能是服务器上的一个**信息查询程序**(例如搜索引擎或动态网页程序)。

在网络上,主机有主机的地址,服务有服务的地址。该如何表示网络资源的地址呢?TCP/IP 使用**统一资源定位符**(Uniform Resource Locator,**URL**)来描述网络资源所在的位置。URL 就是网络资源的地址。

1. 表示数据文件的 URL

如果所访问的网络资源是一个数据文件,则 URL 首先需要指明提供该文件的网络服务地址,然后再指明文件在该服务地址下的路径。例如:

(1) URL“**http**://www.cau.edu.cn:**80**/index.html”所表示的网络资源是网络服务“**http**://www.cau.edu.cn:**80**”下的网页文件“/**index**.html”。

(2) URL“**http**://www.cau.edu.cn/**images**/1182/culture.jpg”所表示的网络资源是网络服务“**http**://www.cau.edu.cn:**80**”下的图像文件“/**images**/1182/culture.jpg”。

如果网络资源是一个数据文件,URL 可以认为是该文件在网络上的文件名。

2. 表示信息查询程序的 URL

网络资源也可能是网络服务提供的一个信息查询程序,例如搜索引擎、JSP/ASP/PHP 动态网页程序、Web Service 服务程序等。如果所访问的网络资源是一个信息查询程序,则网络服务会首先在服务器主机上执行这个程序,然后返回执行所得到的查询结果。表示信息查询程序的 URL 需要指明其网络服务地址、程序路径,另外还可以添加查询条件。例如:

(1) URL“**http**://www.cau.edu.cn/**search**?**id**=1234&**name**=Messi”所表示的网络资源是网络服务“**http**://www.cau.edu.cn:**80**”下的搜索引擎程序“/**search**”,搜索条件是“学号 id 等于 1234,并且姓名 name 等于 Messi”。

(2) URL“**http**://www.cau.edu.cn/**find**.jsp?**id**=1234&**name**=Messi”所表示的网络资源是网络服务“**http**://www.cau.edu.cn:**80**”下的 JSP 动态网页程序“/**find**.jsp”,查询条件是“学号 id 等于 1234,并且姓名 name 等于 Messi”。

如果网络资源是一个信息查询程序,统一资源定位符 URL 可以认为是该程序在网络上的程序文件名。

3. URL 的语法

URL 的语法格式如下：

```
protocol: //host [ : port] path[? query]
```

其中，**protocol**、**host** 和 **port** 描述的是网络服务地址；**path** 描述的是该服务地址下的资源路径；如果资源是一个信息查询程序，则可以添加查询条件 **query**（以问号“?”开头）。

4. 表示本地文件的 URL

可以使用文件名来表示存储在本地计算机硬盘上的某个文件。例如在 Windows 系统中，一个完整的文件名格式如下：

```
盘符:\目录名\子目录名\……\文件名.扩展名
```

也可以使用统一资源定位符 URL 来表示存储在本地硬盘上的文件，其语法格式是在本地文件名之前加上前缀“**file**:///”。例如在 Windows 系统中，本地 D 盘上图像文件“D:\image\1.png”所对应的 URL 是：

```
file:///D:\image\1.png
```

或

```
file:///D: /image /1.png
```

9.2.3 访问网络资源

本节以 Web 服务为例，具体讲解如何编写客户端应用程序来访问网络资源。Web 服务主要用于信息查询，其所提供的网络资源就是一组静态网页文件，或是由信息查询程序自动生成的动态网页。**注**：Web 服务使用的应用层协议是 HTTP，默认端口为 TCP 80。

通常，通过浏览器来访问 Web 服务里的网页。例如，Java 语言官方网站的网址是“**http**://www.**oracle**.com/technetwork/java/**index**.html”。在浏览器中打开这个网址，浏览器将显示该网站的主页，如图 9-8 所示。

Java 语言官方网站的网址“**http**://www.**oracle**.com/technetwork/java/**index**.html”，实际上就是该网站主页的文件名，它是一个 URL 形式的网络文件名。

程序员可以不用浏览器，而是自己编写一个客户端应用程序来访问 Web 服务里的网络文件。其编程过程是：首先使用 Java API 中的统一资源定位符类 **URL** 来存放网络文件的文件名，然后创建该网络文件的输入流对象，这样就可以像读取本地硬盘文件一样来读取网络文件里的数据了。

1. 统一资源定位符类 URL

为了存储、处理统一资源定位符，Java API 提供了一个统一资源定位符类 URL。请读者阅读下面的统一资源定位符类 URL 说明文档。

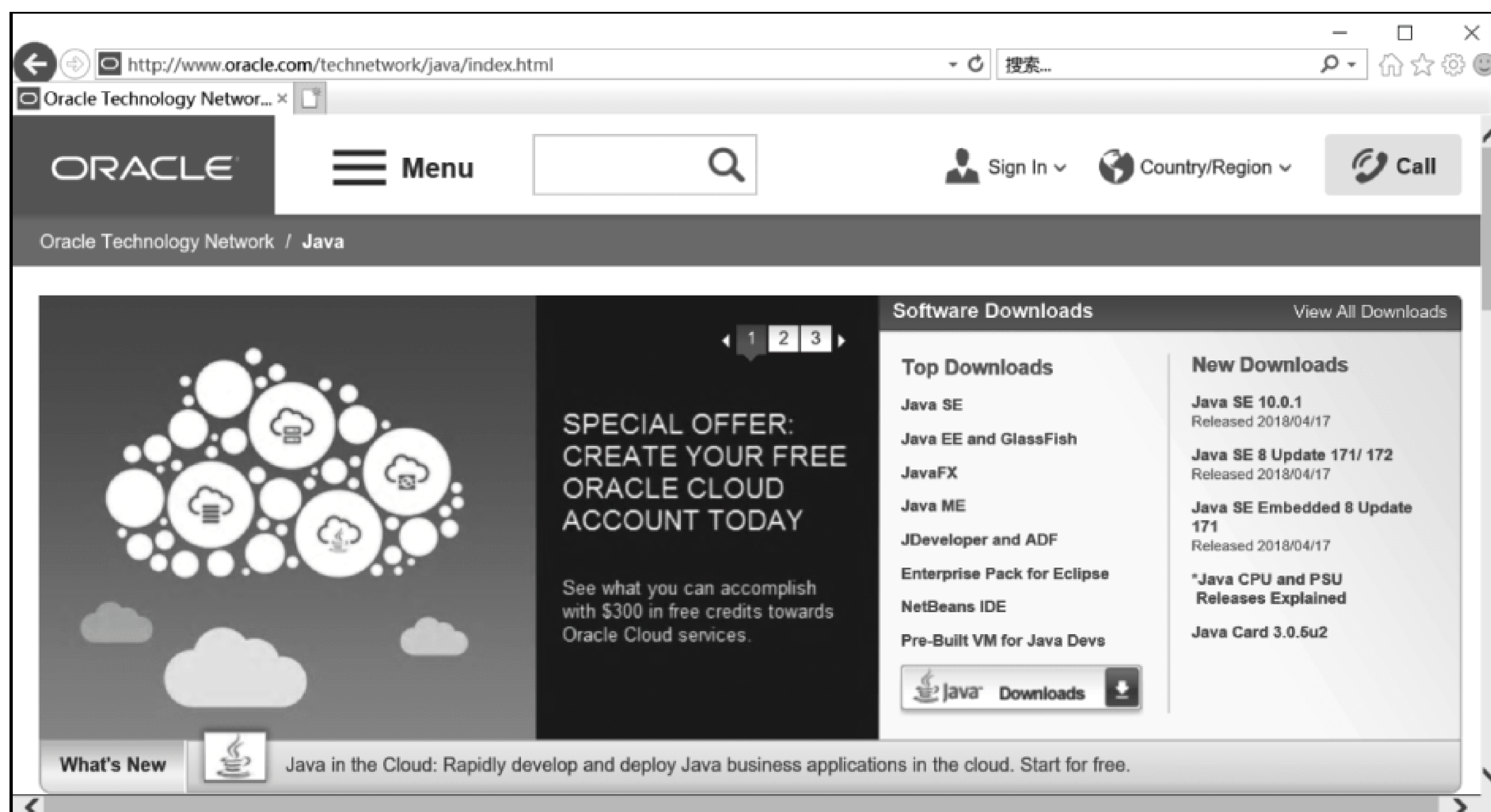


图 9-8　Java 语言官方网站的主页

java. net. **URL** 类说明文档

public final class **URL**
extends **Object**
implements **Serializable**

	修饰符	类成员(节选)	功 能 说 明
1		**URL**(String spec)	构造方法,创建 URL 对象
2		**URL**(String protocol,String host,String file)	构造方法,给定协议、主机名和文件名创建 URL 对象
3		**URL**(String protocol, String host, int port, String file)	构造方法,给定协议、主机名、端口号和文件名创建 URL 对象
4		String **getProtocol**()	获取所使用的服务协议
5		String **getHost**()	获取主机名
6		int **getPort**()	获取端口号
7		int **getDefaultPort**()	获取服务协议的默认端口
8		String **getPath**()	获取资源的存储路径
9		String **getFile**()	获取文件名
10		String **getQuery**()	获取查询参数
11		InputStream **openStream**()	创建 URL 的输入流对象
12		URLConnection **openConnection**()	创建 URL 的连接对象

…

统一资源定位符类 URL 包含了定位网络资源所需的全部信息,其中包括服务器的主机名(或 IP 地址)、服务协议和端口,以及网络资源的存放路径。例 9-2 给出了一个统一资源定位符类 URL 的 Java 演示程序。

例 9-2 一个统一资源定位符类 URL 的 Java 演示程序(JURLTest.java)

```
1  import java.net.*;                          //导入 java.net 网络包中的类
2  import java.io.*;                           //导入 java.io 输入输出流包中的类
3  public class JURLTest {                     //测试类:测试统一资源定位符类 URL 的用法
4      public static void main(String[] args) {                    //主方法
5          try {                               //处理可能出现的勾选异常 MalformedURLException
6              URL url = new URL("http://www.oracle.com/technetwork/java/index.html");
7              System.out.println( "URL:" + url );
8              System.out.println( "协议:" + url.getProtocol() );
9              System.out.println( "主机:" + url.getHost() );
10             System.out.println( "端口:" + url.getPort() );
11             System.out.println( "默认端口:" + url.getDefaultPort() );
12             System.out.println( "路径:" + url.getPath() );
13         }
14         catch(MalformedURLException e) { e.printStackTrace(); } //捕捉并处理勾选异常
15 } }
```

在 Eclipse 集成开发环境中运行例 9-2 的程序,运行结果如图 9-9 所示。

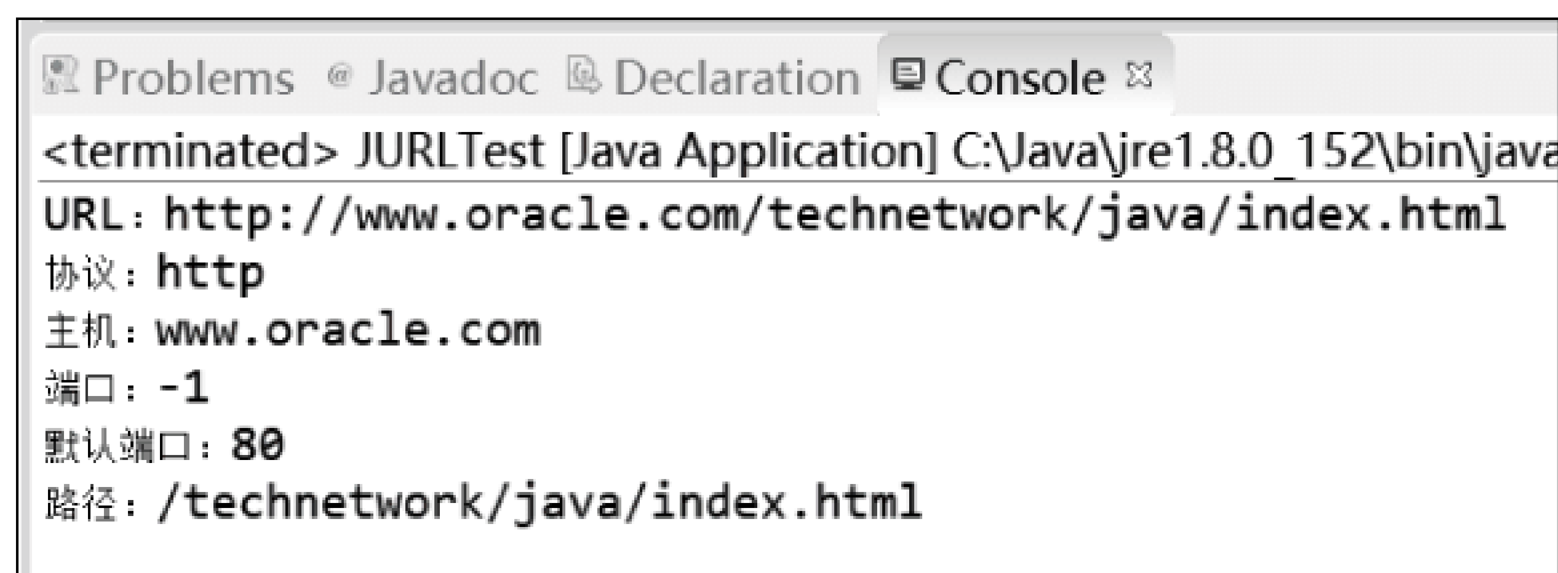


图 9-9 例 9-2 程序的运行结果

2. 编程访问 Web 服务里的网页文件

下面编写一个自己的客户端应用程序来访问 Web 服务里的网页文件,读取其中的网页内容。例 9-3 给出了一个访问 Java 语言网站主页的演示程序。

例 9-3 一个访问 Java 语言网站主页的演示程序(JWebPageTest.java)

```
1  import java.net.*;                          //导入 java.net 网络包中的类
2  import java.io.*;                           //导入 java.io 输入输出流包中的类
3  public class JWebPageTest {                 //测试类:读取网站里的网页文件(html)
4      public static void main(String[] args) { //主方法
5          try {                               //处理可能出现的勾选异常 IOException
6              URL url = new URL("http://www.oracle.com/technetwork/java/index.html");
7              System.out.println("从网页读取信息: " + url);
8              InputStreamReader in = new InputStreamReader( url.openStream() );
9              char cbuf[] = new char[ 300];   //只读 300 个字符
10             int len = in.read(cbuf);
11             for (int n = 0; n < len; n++)
12                 System.out.print( cbuf[n]);
13             System.out.print("......\n 以上是从网页读取的原始信息。");
```

```
14              System.out.println( "字符编码是:" + in.getEncoding() );
15              in.close();
16          }
17          catch(IOException e) { e.printStackTrace(); }  //捕捉并处理勾选异常
18  }  }
```

在 Eclipse 集成开发环境中运行例 9-3 的程序,运行结果如图 9-10 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> JWebPageTest [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe (2018年6月25日 下午4:29:52)
从网页读取信息: http://www.oracle.com/technetwork/java/index.html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtr

<html xmlns="http://www.w3.org/1999/xhtml">
<head><meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <script type="text/javascript">
   var _U = "undefi......
以上是从网页读取的原始信息。字符编码是:GBK
```

图 9-10　Java 语言网站主页的原始信息(前 300 个字符)

图 9-10 显示的 Java 语言网站主页内容是 HTML 格式的原始信息。浏览器程序会在此基础上对 HTML 原始信息做进一步解析,提炼并显示出其中的超文本内容。

3. 编程访问 Web 服务里的图像文件

例 9-4 再给出一个访问 Web 服务里图像文件的 Java 演示程序。

例 9-4　一个访问 Web 服务里图像文件的 Java 演示程序(JWebImageTest.java)

```
 1  import java.net.*;                          //导入 java.net 网络包中的类
 2  import java.io.*;                           //导入 java.io 输入输出流包中的类
 3  import java.awt.*;                          //以下为导入图形用户界面和图像相关的类
 4  import java.awt.image.BufferedImage;
 5  import javax.imageio.ImageIO;
 6  import javax.swing.*;
 7
 8  public class JWebImageTest {                //测试类:加载并显示网站里的图像文件
 9      public static void main(String [] args) { //主方法
10          try {                               //处理可能出现的勾选异常 IOException
11              String netURL = "http://www.cau.edu.cn/images/1182/culture.jpg";
12              System.out.println("从网站读取图片: " + netURL);
13              URL url = new URL(netURL);
14              BufferedImage img = ImageIO.read( url);          //加载网络图像文件
15              //创建框架窗口,显示加载的网络图像
16              JFrame w = new JFrame("显示网络图片");            //创建窗口
17              w.setSize(660,360);  w.setVisible(true);
18              w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19              SwingUtilities.invokeLater( () ->{               //在事件分发线程中绘图
20                  Graphics g = w.getGraphics();                //获取绘图对象
21                  g.setFont( new Font("Times New Rome",0,24) );//设置字体
22                  g.drawString(netURL, 10, 75);                //显示网址
```

```
23                g.drawImage(img, 10, 100, null);           //显示图像
24            } );
25        }
26        catch(IOException e) { e.printStackTrace(); }     //捕捉并处理勾选异常
27  }  }
```

在 Eclipse 集成开发环境中运行例 9-4 的程序，运行结果如图 9-11 所示。



图 9-11　加载并显示中国农业大学网站里的图像文件

本节习题

1. 因特网 Web 服务的默认端口是(　　)。
   A. TCP 80　　B. UDP 80　　C. TCP 21　　D. TCP 25
2. 不能被用作网络上主机地址的是(　　)。
   A. 主机名　　B. 域名　　C. IP 地址　　D. 网络应用程序名
3. 网络服务地址没有包含的内容是(　　)。
   A. 协议　　B. 主机地址　　C. 端口号　　D. 网络资源的文件名
4. 网络资源地址没有包含的内容是(　　)。
   A. 协议　　B. 主机地址　　C. 访问权限　　D. 网络资源的文件名
5. 统一资源定位符类 URL 中创建输入流对象的方法是(　　)。
   A. getProtocol()　　B. getHost()　　C. getFile()　　D. openStream()

9.3　程序之间的网络通信

网络上两台计算机之间的通信，本质上是两个网络应用程序之间的通信。网络应用程序与普通应用程序之间的最大区别在于，网络应用程序具有通信功能，它能与网络上的其他程序互相通信，协同工作。

TCP/IP 网络传输层为网络应用程序提供了两种不同的通信方式，分别是有连接通信(TCP)和无连接通信(UDP)。下面将关注点聚焦到程序间的网络通信上，重点学习如何利用 Java API 中与网络通信相关的类来编写网络应用程序。本节首先讲解基于 TCP 的网络

应用程序，下一节再讲解基于 UDP 的网络应用程序。

9.3.1 TCP 与 Socket

TCP 是一种有连接的通信方式，主要应用于 C/S 架构中客户端应用程序与服务器应用程序之间的通信。其通信流程如下。

(1) **服务器应用程序**首先确定对外服务的 TCP 端口，然后持续**监听**该端口的通信。

(2) **客户端应用程序**向服务器应用程序的服务端口发送**连接请求**，申请在双方之间建立一个 TCP 连接。

(3) **服务器应用程序**接收连接请求，**确认**与客户端应用程序建立 TCP 连接。

(4) 客户端应用程序与服务器应用程序基于所建立的 TCP 连接开始**双向通信**，后续通信内容就是双方所要进行的网络服务的内容。

(5) 服务结束后，通信双方**断开**(关闭)TCP 连接，通信结束。

上述客户端和服务器应用程序之间是基于 TCP 来进行网络服务的“**请求-响应**”(request-response)通信的。为了方便程序员编写这样的程序，Java API 提供了两个基于 TCP 进行网络通信的类，它们分别是套接字类 **Socket** 和服务器套接字类 **ServerSocket**。

1. 套接字

TCP 在一条物理链路上划分出 65536 个端口，不同程序使用不同的端口，这样同一台主机上的多个程序就可以共用一条物理链路进行通信。

想象一下，在物理链路两端各连接着一个通信插座，每个插座上有多个通信插口。每个通信插口相当于是一个 TCP 端口。程序使用某个 TCP 端口进行通信，这类似于是用一根电线将程序插在了某个通信插口上，如图 9-12 所示。

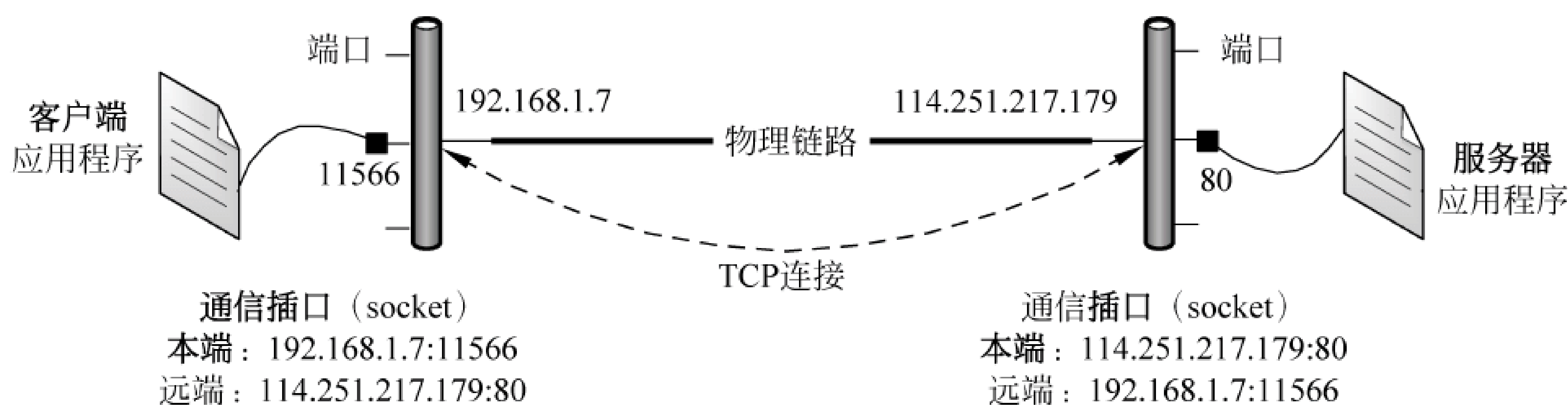


图 9-12 一个 TCP 连接通过两端的通信插口将客户端和服务器应用程序连接在一起

一个 TCP 连接的两端各有一个通信插口，分别连接着客户端应用程序和服务器应用程序。TCP 将通信插口称作 **Socket**，中文译成“**套接字**”。套接字中主要包含**本端**(local)和**远端**(remote)的网络地址和端口信息，它们分别对应了本端和远端的网络应用程序。客户端套接字中的远端指的是服务器，服务器套接字中的远端指的是客户端。

2. Java API 中的套接字类 Socket

为了存储、处理 TCP 连接两端的套接字信息，Java API 提供了一个套接字类 **Socket**。请读者阅读下面的套接字类 Socket 说明文档。

java. net. **Socket** 类说明文档			
public class **Socket** extends **Object** implements **Closeable**			
	修饰符	类成员(节选)	功 能 说 明
1		**Socket**()	构造方法
2		**Socket**(InetAddress address, int port)	构造方法(同时建立连接)
3		**Socket**(String host, int port)	构造方法(同时建立连接)
4		void **connect**(SocketAddress endpoint)	客户端向服务器申请连接
5		void **connect**(SocketAddress endpoint, int timeout)	客户端向服务器申请连接
6		InputStream **getInputStream**()	获取 TCP 连接的字节输入流
7		OutputStream **getOutputStream**()	获取 TCP 连接的字节输出流
8		InetAddress **getInetAddress**()	获取远端的网络地址
9		int **getPort**()	获取远端的端口号
10		InetAddress **getLocalAddress**()	获取本端的网络地址
11		int **getLocalPort**()	获取本端的端口号
12		boolean **isConnected**()	检查是否已建立 TCP 网络连接
13		boolean **isClosed**()	检查 TCP 网络连接是否已断开
14		void **close**()	断开 TCP 网络连接
…			

客户端应用程序使用套接字类 Socket 可以很方便地连接网络上的服务器。例 9-5 给出了一个使用套接字类 Socket 连接中国农业大学 Web 服务器的 Java 演示程序。**注**:中国农业大学 Web 服务器的域名是 www.cau.edu.cn,服务端口为 TCP 80。

例 9-5 使用套接字类 Socket 连接 Web 服务器的 Java 演示程序(JSocketTest.java)

```
 1  import java.net.*;                             //导入 java.net 网络包中的类
 2  import java.io.*;                              //导入 java.io 输入输出流包中的类
 3
 4  public class JSocketTest {                     //测试类:测试套接字类 Socket 的用法
 5      public static void main(String[] args) {  //主方法
 6          try {                                  //处理可能出现的勾选异常
 7              //创建套接字对象,向服务器申请建立 TCP 连接
 8              Socket s = new Socket("www.cau.edu.cn", 80);  //给出服务器的域名和端口
 9              System.out.println(s + "connected......");
10              //显示套接字中的本端信息
11              System.out.println( "local:" + s.getLocalAddress() + ":" + s.getLocalPort() );
12              //显示套接字中的远端信息
13              System.out.println( "remote: " + s.getInetAddress() + ":" + s.getPort() );
14              //建立 TCP 连接后可以与服务器进行通信,本例暂不通信
15              s.close();                                    //断开 TCP 网络连接
16              System.out.println("TCP connection closed......");
```

```
17            }
18            catch(UnknownHostException e) { System.out.println("Host not found"); }
19            catch(ConnectException e) { System.out.println("Host connection failed"); }
20            catch(IOException e) { System.out.println("IOException"); }
21    }  }
```

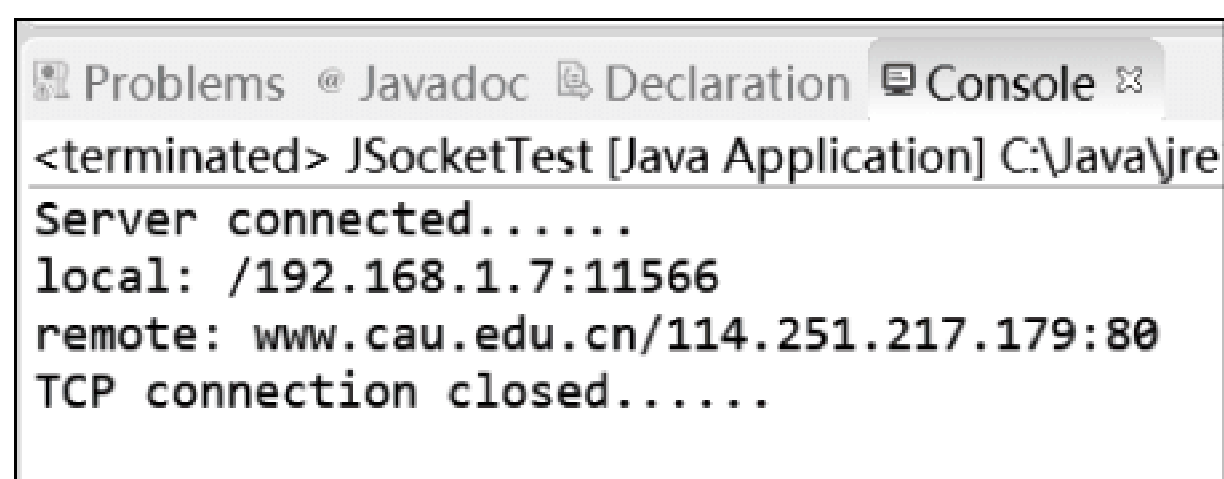


图 9-13 例 9-5 程序的运行结果

在作者的计算机上运行例 9-5 的程序，运行结果如图 9-13 所示。

图 9-13 中 TCP 连接的详细连接示意图可参见前面的图 9-12。客户端应用程序向服务器应用程序申请建立 TCP 连接，连接时服务器端的 TCP 端口是固定不变的，而客户端的 TCP 端口是随机分配的，每次连接可能都不一样。

9.3.2 C/S 架构程序的代码框架

本节讲解基于 TCP 编写 C/S 架构程序的代码框架，其中包括客户端和服务器两个应用程序的代码结构。

1. 客户端应用程序的代码结构

在 C/S 架构中，客户端应用程序是使用网络服务的。其主要算法流程是：向服务器申请建立 TCP 连接；连接成功后与服务器进行通信，发送服务请求，然后接收服务响应；服务结束后断开 TCP 连接。客户端应用程序应当具有如下代码结构。

```
try { //处理可能出现的勾选异常
    //创建套接字对象,向服务器申请建立 TCP 连接
    Socket s = new Socket(服务器域名或 IP 地址, 服务器端口);
    ......        //连接成功后与服务器进行通信,发送服务请求,然后接收服务响应
    s.close();    //服务结束后断开 TCP 连接
}
catch(IOException e) { System.out.println("IOException"); }
```

9.3.1 节的例 9-5 就是一个具有上述代码结构的客户端应用程序。

2. 服务器应用程序的代码结构

在 C/S 架构中，服务器应用程序是提供网络服务的。其主要算法流程是：首先确定对外服务的 TCP 端口，然后持续**监听**(listen)该端口的通信；**接收**(accept)客户端发来的连接请求，确认建立 TCP 连接；连接成功后与客户端进行通信，接收服务请求，然后发送服务响应；服务结束后断开 TCP 连接。

服务器应用程序如何监听某个 TCP 端口，如何接收客户端的 TCP 连接请求并确认建立连接呢？这就需要用到 Java API 中的服务器套接字类 **ServerSocket**。请读者阅读下面的服务器套接字类 ServerSocket 说明文档。

java.net.**ServerSocket** 类说明文档			
public class **ServerSocket** extends **Object** implements **Closeable**			
	修饰符	类成员(节选)	功 能 说 明
1		**ServerSocket**()	构造方法
2		**ServerSocket**(int port)	构造方法(同时指定监听端口)
3		**ServerSocket**(int port,int backlog, InetAddress bindAddr)	构造方法(同时指定监听端口等)
4		Socket **accept**()	监听服务请求。如果有请求,则建立 TCP 连接,返回套接字,否则保持阻塞状态
5		int **getLocalPort**()	获取所监听的端口
6		InetAddress **getInetAddress**()	获取本服务的网络地址
7		void **bind**(SocketAddress endpoint)	将服务绑定到指定的套接字地址
8		SocketAddress **getLocalSocketAddress**()	获取本服务的套接字地址
9		boolean **isBound**()	检查是否已绑定地址和端口
10		boolean **isClosed**()	检查服务是否已关闭
11		void **close**()	关闭网络服务
…			

服务器应用程序应当具有如下代码结构。

```
try { //处理可能出现的勾选异常
    //创建服务器套接字对象,用于监听某个 TCP 端口
    ServerSocket ss = new ServerSocket(服务端口);
    while (运行条件或 true) {      //服务器应保持运行状态,随时接收客户端的连接请求
        Socket s = ss.accept();  //如果有 TCP 连接请求,则确认建立连接,返回套接字对象
        ……                      //连接成功后与客户端进行通信,接收服务请求,然后发送服务响应
        s.close();               //服务结束后断开 TCP 连接
        //继续循环,准备接收下一个连接请求。可接收不同客户端的连接请求
    }
    ss.close();                  //关闭网络服务
}
catch(IOException e) { System.out.println("IOException"); }
```

3. 套接字对象的输入输出流

在 TCP 连接成功后,客户端与服务器应用程序各有一个套接字 socket 的对象,它们分别表示 TCP 连接两端的通信插口。Java API 将程序从通信插口(即 socket 对象)**接收数据**抽象成**输入流**,向通信插口**发送数据**抽象成**输出流**,这样网络通信问题就被转换成了输入输出问题。套接字类 Socket 中有如下两个重要方法。

(1) **getInputStream**()。获得套接字的字节型输入流对象。从输入流对象读取数据,就是接收对方发来的信息。

(2) **getOutputStream**()。获得套接字的字节型输出流对象。向输出流对象写入数据,

就是向对方发送信息。

可以根据需要将字节型输入输出流对象包装成其他流对象,例如包装成字符型流对象、带缓冲区的流对象等。关于输入输出流,请见第7章。

Java API的套接字类Socket和服务器套接字类ServerSocket封装了TCP网络通信协议的所有实现细节,程序员只要使用这两个类就可以很容易地编写出各种网络应用程序。图9-14给出了网络编程与计算机网络之间的关系示意图。

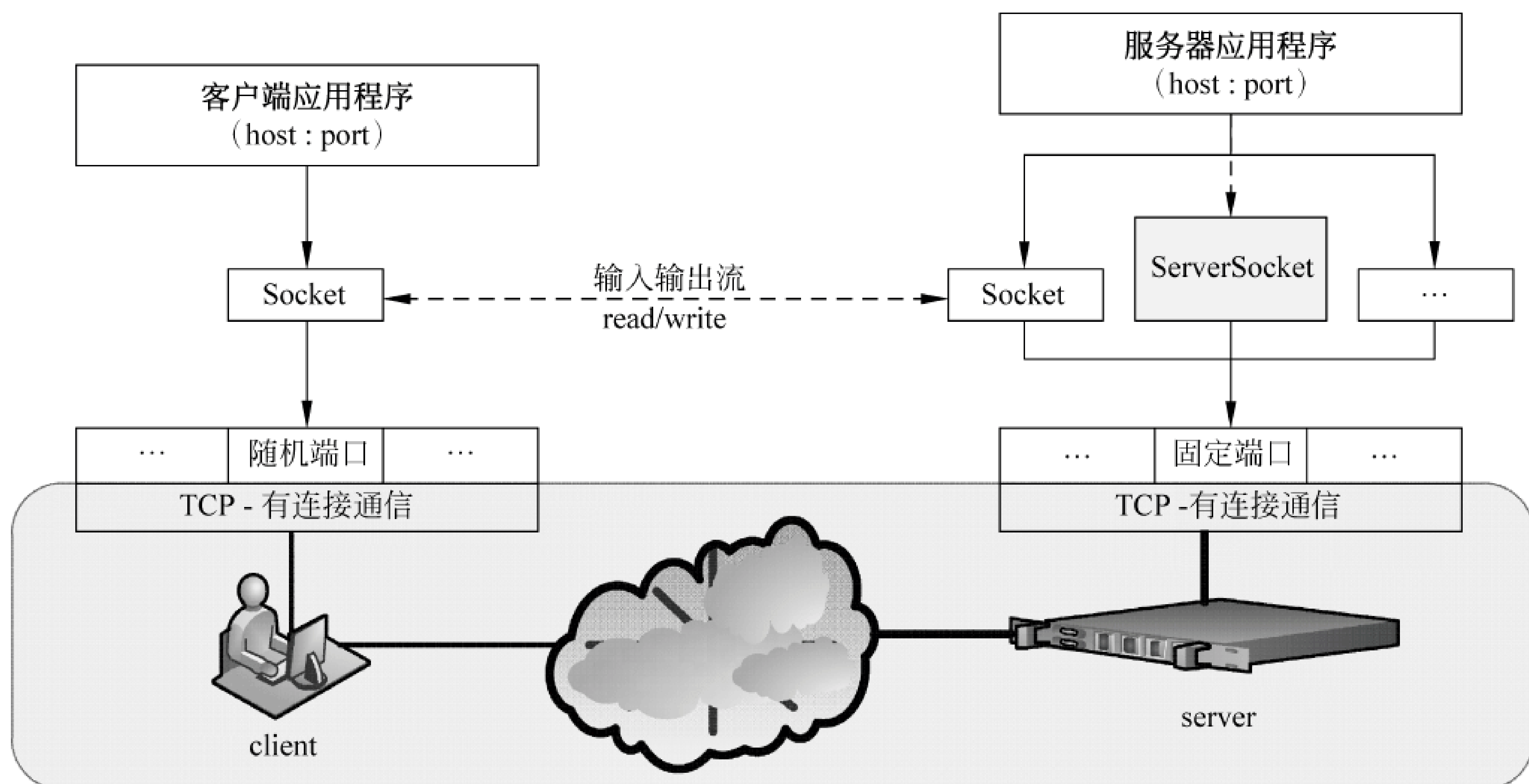


图9-14 网络编程与计算机网络之间的关系示意图

在图9-14中,网络应用程序之间的通信是通过Socket、ServerSocket这两个Java API类实现的。编写网络应用程序,程序员并不需要完全了解计算机网络的内部细节,只需要了解其基本工作原理即可。

9.3.3 C/S架构演示程序

本节给出一个完整的C/S架构**时间服务**演示程序。其中的服务器应用程序提供一个标准时间服务,而客户端应用程序则是使用该服务,查询标准时间。

假设程序员首先为时间服务设计一个如下的服务规范。

(1) **客户端**向服务器发送服务请求,请求时需提交客户名称。假设时间服务**约定**:所提交的客户名称必须为7个字符长度,例如Client1、Client2等。

(2) **服务器**接收服务请求,然后响应请求,提供时间服务。假设时间服务**约定**:返回给客户端的响应信息中应当包含对客户的问候,然后再给出标准时间。例如,返回给客户端Client1的响应信息应当是"Hello, Client1! 标准时间"。

(3) **客户端**接收服务器返回的响应信息,服务结束。

上述规范约定了时间服务的内容和流程,这个规范就是一个关于时间服务的应用层协议。程序员应当严格按照这个协议,分别编写一个提供时间服务的服务器应用程序和一个使用时间服务的客户端应用程序。

为便于调试，程序员可以在自己的计算机上同时运行客户端和服务器应用程序。TCP/IP 网络规定本地主机的主机名为 **localhost**，IP 地址为 **127.0.0.1**。客户端应用程序在访问本地主机上的网络服务时，可以直接使用这个特殊的主机名或 IP 地址。

1. 使用时间服务的客户端应用程序

假设时间服务程序中的客户端和服务器应用程序运行在同一台主机上，所使用的服务端口为 TCP 8000。例 9-6 首先给出使用时间服务的 Java 客户端应用程序示例代码。

例 9-6 使用时间服务的 Java 客户端应用程序示例代码(JTimeClient.java)

```
1   import java.net.*;                                  //导入 java.net 网络包中的类
2   import java.io.*;                                   //导入 java.io 输入输出流包中的类
3   public class JTimeClient {                          //主类:使用时间服务的客户端应用程序
4       public static void main(String[] args) {        //主方法
5           //客户端算法流程:发送 TCP 连接请求,然后发送服务请求,接收服务响应
6           Socket s = null;                            //套接字
7           InputStreamReader in = null;                //用于接收信息的输入流
8           OutputStreamWriter out = null;              //用于发送信息的输出流
9           try {                                       //处理可能出现的勾选异常
10              //创建套接字对象,申请建立与本机上服务器应用程序的 TCP 连接
11              s = new Socket("localhost", 8000);      //服务端口:8000
12              System.out.println("localhost:8000 connected......");
13              //发送服务请求:获取套接字的字节型输出流,然后包装成字符型
14              out = new OutputStreamWriter( s.getOutputStream() );
15              int id = (int)(Math.random() * 10); //随机生成一个 0～9 的用户号
16              String request = "Client" + id; //约定用户名是 7 个字符长度
17              out.write(request, 0, request.length());      //向服务器发送服务请求
18              out.flush();                            //立即发送
19              System.out.println("发送的服务请求是: " + request);
20              //接收服务响应:获取套接字的字节型输入流,然后包装成字符型
21              in = new InputStreamReader( s.getInputStream() );
22              char[] buf = new char[100];             //用于存储接收到的响应(最多 100 个字符)
23              in.read(buf, 0, buf.length);            //读取响应,例如:Hello, Client1! 标准时间
24              String response = new String(buf);//将字符数组包装成字符串
25              System.out.print("接收的服务响应是: " + response);
26              System.out.println("   其编码是: " + in.getEncoding());
27          } catch(IOException e) { System.out.println("IOException"); }
28          finally {                                   //服务结束后关闭输入流、输出流以及 TCP 连接
29              try {                                   //处理可能出现的勾选异常
30                  if (in != null)   in.close();       //关闭接收信息的输入流
31                  if (out != null) out.close();       //关闭发送信息的输出流
32                  if (s != null)    s.close();        //断开 TCP 网络连接
33              } catch(IOException e) { System.out.println("IOException"); }
34          }
35  }  }
```

2. 提供时间服务的服务器应用程序

例 9-7 给出了提供时间服务的 Java 服务器应用程序示例代码。

例 9-7 提供时间服务的 Java 服务器应用程序示例代码(JTimeServerST.java)

```
1   import java.net.*;                    //导入 java.net 网络包中的类
2   import java.io.*;                     //导入 java.io 输入输出流包中的类
3   import java.time.*;                   //导入 java.time 包中与时间相关的类
4
5   public class JTimeServerST {          //主类:提供时间服务的服务器应用程序
6       public static void main(String[] args) {//主方法
7           //服务器算法流程:监听服务端口,接收 TCP 连接请求,然后进行服务请求 - 响应
8           try {                         //处理可能出现的勾选异常
9               //创建服务器套接字对象,用于监听某个 TCP 端口
10              ServerSocket ss = new ServerSocket(8000);     //监听 TCP 8000 端口
11              System.out.println("HelloServer started at 8000 ......");
12              while (true) {            //服务器应一直处于运行状态,随时处理服务请求
13                  Socket s = ss.accept(); //监听连接请求,等待与客户端建立 TCP 连接
14                  System.out.print("\n 收到一个服务请求,并建立了 TCP 连接: ");
15                  System.out.println( s.getInetAddress() + ":" + s.getPort() );
16                  timeService(s);       //执行服务请求 - 响应算法
17                  //继续循环,监听下一个服务请求。可接收不同客户端的连接请求
18              }
19          } catch(IOException e) { System.out.println("IOException"); }
20      }
21
22      public static void timeService(Socket s) {//请求 - 响应:接收客户名,然后返回时间信息
23          InputStreamReader in = null;      //用于接收信息的输入流
24          OutputStreamWriter out = null;    //用于发送信息的输出流
25          try {                         //处理可能出现的勾选异常
26              //接收服务请求:获取套接字的字节型输入流,然后包装成字符型
27              in = new InputStreamReader( s.getInputStream() );
28              char[] buf = new char[ 7];    //用于存储客户名(约定是 7 个字符长度)
29              in.read(buf, 0, buf.length);  //读取客户名,例如"Client1"
30              String request = new String(buf);    //将字符数组包装成字符串
31              System.out.print("客户端的服务请求是: " + request);
32              System.out.println("   其编码是: " + in.getEncoding());
33              //发送服务响应:获取套接字的字节型输出流,然后包装成字符型
34              out = new OutputStreamWriter( s.getOutputStream() );
35              LocalDateTime t = LocalDateTime.now();    //假设本机时间为标准时间
36              //生成响应信息,例如"Hello, Client1! 标准时间",然后发送给客户端
37              String response = String.format("Hello, %s! %s", request, t );
38              out.write(response, 0, response.length()); //向客户端发送响应信息
39              out.flush();                              //立即发送
40              System.out.println("返回客户端的响应是: " + response);
41          } catch(IOException e) { System.out.println("IOException"); }
42          finally {                         //服务结束后关闭输入流、输出流以及 TCP 连接
43              try {                         //处理可能出现的勾选异常
44                  if (in != null)  in.close(); //关闭接收信息的输入流
```

```
45                  if (out != null) out.close();  //关闭发送信息的输出流
46                  if (s != null)   s.close();    //断开 TCP 网络连接
47              } catch(IOException e) { System.out.println("IOException"); }
48          }
49          //至此已执行完一轮时间服务的请求－响应算法
50  }  }
```

在 Eclipse 集成开发环境中首先运行例 9-7 的服务器应用程序，启动时间服务；然后再运行例 9-6 的客户端应用程序，使用时间服务。两个程序的运行结果如图 9-15 所示。

```
Problems  Javadoc  Declaration  Console
<terminated> JTimeClient [Java Application] C:\Java\jre1.8.0_152\bin
localhost:8000 connected......
发送的服务请求是: Client1
接收的服务响应是: Hello, Client1! 2018-06-28T14:57:18.567
```

(a) 例 9-6 客户端应用程序的运行结果

```
Problems  Javadoc  Declaration  Console
JTimeServerST [Java Application] C:\Java\jre1.8.0_152\bin\javaw.exe (2018
TimeServer started at 8000 ......

收到一个服务请求，并建立了TCP连接: /127.0.0.1:14863
客户端的服务请求是: Client1  其编码是: GBK
返回客户端的响应是: Hello, Client1! 2018-06-28T14:57:18.567
```

(b) 例 9-7 服务器应用程序的运行结果

图 9-15　例 9-6 和例 9-7 时间服务程序的运行结果

图 9-15(a)显示了客户端应用程序连接时间服务器，使用时间服务的请求-响应过程。图 9-15(b)显示了服务器应用程序启动时间服务，然后接收客户端请求并返回标准时间的过程。其中，图 9-15(b)倒数第二行的最后显示了一个 GBK，它表示客户端和服务器之间信息通信所采用的字符编码是 GBK 编码。GBK 编码是中文 Windows 操作系统的默认编码。可以在将字节型输入输出流包装成字符型时指定其他字符编码，例如可以按如下形式来指定 UTF-8 编码：

```
in  = new InputStreamReader(s.getInputStream(), "UTF - 8");
out = new OutputStreamWriter(s.getOutputStream(), "UTF - 8");
```

在服务器应用程序运行过程中，客户端应用程序可以多次运行，每运行一次即完成一次时间服务的请求-响应过程。客户端应用程序也可以在多台计算机上运行，这相当于是多个用户在同时使用时间服务。

例 9-7 所示的服务器应用程序是一个单线程（即主线程）串行程序。当有多个客户请求时，主线程在同一时刻只能处理一个客户请求，其他客户请求需串行等待。

3. 将例 9-7 改写成多线程并发服务程序

C/S 架构中的服务器应用程序应当采用多线程编程，这样可以提高服务效率。将例 9-7 的服务器应用程序由单线程串行修改成多线程并发，其修改过程只需两步即可完成。

1) 将服务请求-响应算法包装成可在线程中运行的算法对象

将服务请求-响应算法(例 9-7 中代码第 23～48 行)包装成可在线程中运行的算法对象,即实现 Runnable 接口,然后在抽象方法 run()中编写服务请求-响应的算法代码。

2) 在线程中运行算法对象

服务器应用程序在接收到客户端的连接请求后,不是直接在主线程中运行服务请求-响应算法,而是另外创建子线程并在子线程中运行包装后的服务请求-响应算法对象。

例 9-8 给出了修改后的 Java 多线程服务器应用程序示例代码。

例 9-8 提供时间服务的 Java 多线程服务器应用程序示例代码(JTimeServerMT.java)

```
1  import java.net.*;                          //导入 java.net 网络包中的类
2  import java.io.*;                           //导入 java.io 输入输出流包中的类
3  import java.time.*;                         //导入 java.time 包中与时间相关的类
4  public class JTimeServerMT {                //主类:提供时间服务的多线程服务器应用程序
5      public static void main(String[] args) {  //主方法
6          //服务器算法流程:监听服务端口,接收 TCP 连接请求,然后进行服务请求 - 响应
7          try {                               //处理可能出现的勾选异常
8              //创建服务器套接字对象,用于监听某个 TCP 端口
9              ServerSocket ss = new ServerSocket(8000);    //监听 TCP 8000 端口
10             System.out.println("HelloServer started at 8000 ......");
11             while (true) {                  //服务器应一直处于运行状态,随时处理服务请求
12                 Socket s = ss.accept();  //监听连接请求,等待与客户端建立 TCP 连接
13                 System.out.print("\n 收到一个服务请求,并建立了 TCP 连接: ");
14                 System.out.println( s.getInetAddress() +":" +s.getPort() );
15                 //timeService(s);           //执行服务请求 - 响应算法
16                 //另外创建子线程并在子线程中运行时间服务的请求 - 响应算法
17                 TimeService a = new TimeService(s);     //创建 TimeService 算法对象
18                 Thread t = new Thread(a);//在线程中运行算法对象
19                 t.start();                  //启动线程
20                 //继续循环,监听下一个服务请求。可并发接收不同客户端的连接请求
21             }
22         } catch(IOException e) { System.out.println("IOException"); }
23 }  }
24
25 class TimeService implements Runnable {     //时间服务算法类:接收客户名,返回时间信息
26     private Socket s = null;                //与用户连接的 Socket
27     TimeService(Socket userSocket) {  s = userSocket;  }    //构造方法
28     public void run() {             //实现抽象方法 run(),编写服务请求 - 响应的算法代码
29         …                                   //与例 9 - 7 中代码第 23～48 行相同,此处省略
30         //至此已执行完时间服务的请求 - 响应算法,线程将随即结束
31 }  }
```

请读者重点关注例 9-8 中的两段代码:代码第 17～19 行演示了如何创建子线程并在子线程中运行网络服务的请求-响应算法,代码第 25～31 行演示了如何将服务请求-响应算法包装成可在线程中运行的算法类。

本节最后对基于 TCP 的 C/S 架构网络服务程序做一个总结。基于 TCP 的 C/S 架构

网络服务包括两个程序：一个是提供服务的服务器应用程序；另一个是使用服务的客户端应用程序。这两个程序都是基于 TCP 来进行网络服务的"请求-响应"通信的，图 9-16 给出了它们之间的通信流程及代码框架。编写如图 9-16 所示的 C/S 架构网络服务程序，程序员需要用到 Java API 中的套接字类 Socket 和服务器套接字类 ServerSocket。

Client

Server

客户端应用程序的代码框架

① 发起TCP连接，例如：
```
Socket s = new Socket("localhost", 8000);
```

② 发送服务请求，例如：
```
OutputStream out = s.getOutputStream();
out.write( ...... );
```

③ 接收服务响应，例如：
```
InputStream in = s.getInputStream();
in.read( ...... );
```

④ 断开TCP连接，服务结束。
```
s.close();
```

建立TCP连接

服务请求

服务响应

断开TCP连接

服务器应用程序的代码框架

创建服务，监听TCP端口，例如：
```
ServerSocket ss = new ServerSocket(8000);
```

① 接收TCP连接请求，确认连接
```
Socket s = ss.accept();
```

② 接收服务请求，例如：
```
inputStream in = s.getInputStream();
in.read( ...... );
```

③ 发送服务响应，例如：
```
outputStream out = s.getOutputStream();
out.write( ...... );
```

④ 断开TCP连接。
```
s.close();
```

返回①，继续监听连接请求

图 9-16 基于 TCP 的 C/S 架构网络服务程序的通信流程及代码框架

本节习题

1. 一个 TCP 连接包含(　　)个套接字。
   A. 0　　B. 1　　C. 2　　D. 4
2. 套接字中没有包含的信息是(　　)。
   A. 本端 IP 地址　　B. 本端端口号
   C. 远端 IP 地址　　D. 应用层协议名称
3. 套接字类 Socket 中向服务器申请建立 TCP 连接的方法是(　　)。
   A. connect()　　B. getInputStream()
   C. getOutputStream()　　D. close()
4. 套接字类 Socket 中获取 TCP 连接输入流的方法是(　　)。
   A. connect()　　B. getInputStream()

C. getOutputStream()　　D. close()

5. 服务器套接字类 ServerSocket 中接收并确认客户端 TCP 连接请求的方法是(　　)。

A. connect()　　B. accept()

C. listen()　　D. getInetAddress()

6. 下列关于 TCP 的描述中,错误的是(　　)。

A. TCP 是有连接的通信　　B. TCP 可以实现双向通信

C. TCP 必须先连接再通信　　D. TCP 不能实现单向通信

7. 下列关于 C/S 架构网络服务应用程序的描述中,错误的是(　　)。

A. C/S 架构中可以一个服务器对一个客户端

B. C/S 架构中可以一个服务器对多个客户端

C. C/S 架构中服务器的 TCP 端口是固定不变的

D. C/S 架构中客户端的 TCP 端口是固定不变的

8. 下列关于 C/S 架构网络服务应用程序的描述中,错误的是(　　)。

A. C/S 架构中服务器应用程序应当一直保持运行状态

B. C/S 架构中客户端应用程序应当一直保持运行状态

C. C/S 架构中客户端与服务器之间的 TCP 连接是由客户端发起的

D. C/S 架构中服务器需要监听并确认客户端的 TCP 连接请求

9.4 基于 UDP 的网络通信

TCP(传输控制协议)是一种有连接的**双向**通信协议,而 UDP(用户数据报协议)是一种无连接的**单向通信**协议。本节讲解基于 UDP 的网络通信程序。

9.4.1 基于 UDP 通信程序的代码框架

UDP 提供了一种无连接的单向通信方式。如果两个网络应用程序之间使用 UDP 传输数据,则其中一个是**发送方**(sender),另一个是**接收方**(receiver)。发送方与接收方之间的通信流程如下。

(1) **接收方**应用程序首先准备好接收数据的缓冲区,然后监听某个 UDP 端口,等待接收该端口传输过来的数据。

(2) **发送方**应用程序首先准备好接收方的网址、UDP 端口和需要发送的数据,然后向接收方发送数据,通信结束。发送方不需要等待接收方的回复。

(3) **接收方**应用程序接收数据,然后分析并处理数据,通信结束。接收方不需要回复发送方。

上述发送方和接收方应用程序之间是基于 UDP 来进行数据的"**发送-接收**"(send-receive)通信的。为了方便程序员编写这样的程序,Java API 提供了两个基于 UDP 进行网络通信的类,它们分别是数据报包裹类 **DatagramPacket** 和数据报套接字类 **DatagramSocket**。

1. 数据报包裹类 DatagramPacket

顾名思义，数据报包裹类 DatagramPacket 描述的是一个内含数据的包裹。发送方使用数据报包裹类 DatagramPacket 来存放即将被发送的数据，接收方同样使用这个类来存放接收到的数据。请读者阅读下面的数据报包裹类 DatagramPacket 说明文档。

java.net.**DatagramPacket** 类说明文档

public final class **DatagramPacket**
extends **Object**

	修饰符	类成员(节选)	功 能 说 明
1		**DatagramPacket**(byte[] buf, int length)	构造方法
2		**DatagramPacket**(byte[] buf, int length, InetAddress address, int port)	构造方法，指定接收方的网络地址、UDP 端口
3		InetAddress **getAddress**()	获取对方的网络地址
4		int **getPort**()	获取对方的 UDP 端口号
5		int **getLength**()	获取数据报的数据长度
6		byte[] **getData**()	读取数据报包裹中的数据
7		void **setAddress**(InetAddress iaddr)	设置接收方的网络地址
8		void **setPort**(int iport)	设置接收方的端口号
9		void **setLength**(int length)	设置即将被发送的数据长度
10		void **setData**(byte[] buf)	设置将即将被发送的数据

…

2. 数据报套接字类 DatagramSocket

发送方使用数据报套接字类 DatagramSocket 来发送数据，接收方同样使用这个类来接收数据。请读者阅读下面的数据报套接字类 DatagramSocket 说明文档。

java.net.**DatagramSocket** 类说明文档

public class **DatagramSocket**
extends **Object**
implements **Closeable**

	修饰符	类成员(节选)	功 能 说 明
1		**DatagramSocket**()	构造方法
2		**DatagramSocket**(int port)	构造方法，指定本端的 UDP 端口
3		**DatagramSocket**(int port, InetAddress laddr)	构造方法，指定本端的端口和网络地址
4		void **send**(DatagramPacket p)	发送数据报包裹
5		void **receive**(DatagramPacket p)	接收数据报包裹
6		void **setSoTimeout**(int timeout)	设置数据报包裹的有效时间
7		InetAddress **getLocalAddress**()	获取本端网络地址
8		int **getLocalPort**()	获取本端端口号
9		void **close**()	关闭套接字

…

图 9-17 给出了基于 UDP 网络应用程序的通信流程及代码框架。

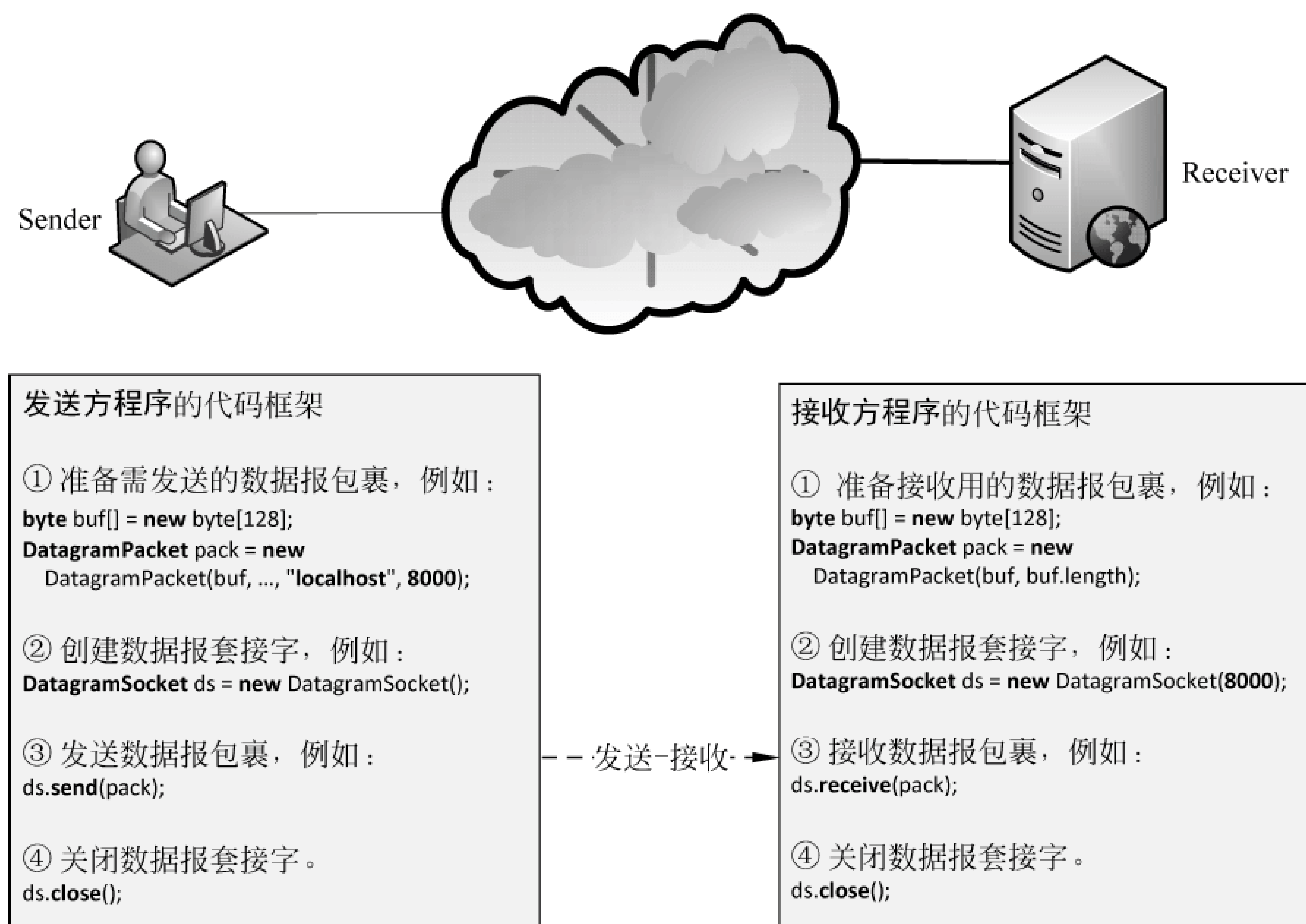


图 9-17 基于 UDP 网络应用程序的通信流程及代码框架

通过图 9-17 所示的 UDP 通信流程可以看出，UDP 发送方在发送完数据之后就结束通信了。它并没有等待接收方的回复，无法确认对方是否收到了数据，因此 UDP 不是一种百分之百可靠的数据传输方式。

接收方应用程序也可以在接收到数据之后向发送方回复一个信息，但这属于是它们之间互换发送方/接收方身份，又开始了下一次 UDP 通信过程。

3. 一个 UDP 通信演示程序

这里给出一个完整的 UDP 通信演示程序，其中包括发送方和接收方两个应用程序。为便于调试，程序员可以在同一台计算机主机上运行这两个程序。接收方应用程序需要先运行，等待接收数据；然后再运行发送方应用程序，向本机(localhost)上的接收方发送一条"Hello，World!"信息。例 9-9 和例 9-10 分别给出了接收方和发送方应用程序的 Java 示例代码。

例 9-9 基于 UDP 的接收方应用程序示例代码(JUDPReceiver.java)

```
1  import java.net.*;                              //导入 java.net 网络包中的类
2  import java.io.*;                               //导入 java.io 输入输出流包中的类
3  public class JUDPReceiver {                     //主类:UDP 协议接收方的演示程序
4      public static void main(String[] args) {    //主方法
5          try {                                   //处理可能出现的勾选异常
6              System.out.println("Receive data at 8000.....\n");
7              //接收前的准备工作:准备好存储缓冲区,将其包装成数据报包裹对象
8              byte buf[] = new byte[128];         //创建一个数据缓冲区(最多接收 128 字节)
```

```
 9                DatagramPacket pack = new DatagramPacket(buf, buf.length);
10                //创建数据报套接字对象,监听某个 UDP 端口,等待接收数据报包裹
11                DatagramSocket ds = new DatagramSocket(8000);     //监听 UDP 8000 端口
12                ds.receive(pack);                                 //接收数据报包裹
13                //分析接收到的数据报包裹,例如发送方的网址、端口和数据等
14                InetAddress udpSender = pack.getAddress();        //获取发送方的网络地址
15                int port = pack.getPort();                        //获取发送方的端口
16                //数据报里的数据是字节数组,可将其转成字符串
17                String msg = new String( pack.getData(), 0, pack.getLength() ); //转字符串
18                System.out.println("Receive data from " + udpSender + ":" + port);
19                System.out.println("所接收到的数据:" + msg);
20                ds.close();                                       //关闭数据报套接字
21            } catch(IOException e){ System.out.println( e.getMessage() ); }
22    }  }
```

例 9-10 基于 UDP 的发送方应用程序示例代码(JUDPSender.java)

```
 1  import java.net.*;                              //导入 java.net 网络包中的类
 2  import java.io.*;                               //导入 java.io 输入输出流包中的类
 3  public class JUDPSender {                       //主类:UDP 协议发送方的演示程序
 4      public static void main(String[] args) {    //主方法
 5          try {                                   //处理可能出现的勾选异常
 6              System.out.print("Send data to localhost:8000...... ");
 7              //发送前的准备工作:准备好接收方网址、端口和需发送的数据
 8              InetAddress udpReceiver = InetAddress.getByName("localhost"); //发给本机
 9              int port = 8000;                    //接收方端口(本例为 UDP 8000)
10              String msg = "Hello, World!";       //将被发送的信息(本例为"Hello, World!")
11              byte buf[] = msg.getBytes();        //将字符串信息转成字节数组
12              //创建数据报包裹对象,其中包含需被发送的数据、接收方的网址和端口
13              DatagramPacket pack = new DatagramPacket(buf, buf.length, udpReceiver, port);
14              //创建数据报套接字对象,然后发送准备好的数据报包裹对象 pack
15              DatagramSocket ds = new DatagramSocket();  //创建一个数据报套接字对象
16              ds.send(pack);                             //发送数据报包裹
17              ds.close();                                //关闭数据报套接字
18              System.out.println("Done");
19          } catch(IOException e) { System.out.println( e.getMessage() ); }
20  }  }
```

在 Eclipse 集成开发环境中首先运行例 9-9 的接收方应用程序,然后再运行例 9-10 的发送方应用程序,向接收方发送一条"Hello, World!"信息。图 9-18 给出了例 9-9 接收方应用程序的运行结果。

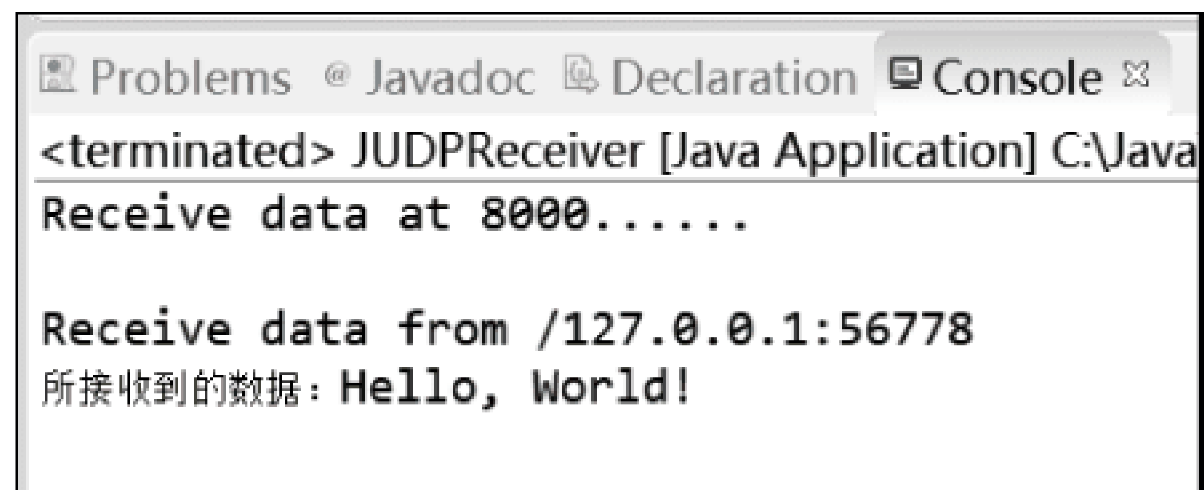


图 9-18 例 9-9 接收方应用程序的运行结果

9.4.2 UDP 接收服务器

设想这样一种编程场景,多个数据采集点需要将采集到的数据集中上传到某个数据服务器上。可以在每个采集点安装一个发送方应用程序,然后通过 UDP 将采集数据上传给同一个接收方应用程序,即同一台数据服务器。在这个编程场景中,参与 UDP 通信的有多个发送方,但只有一个接收方,这个接收方称为 **UDP 接收服务器**(UDP Server),如图 9-19 所示。

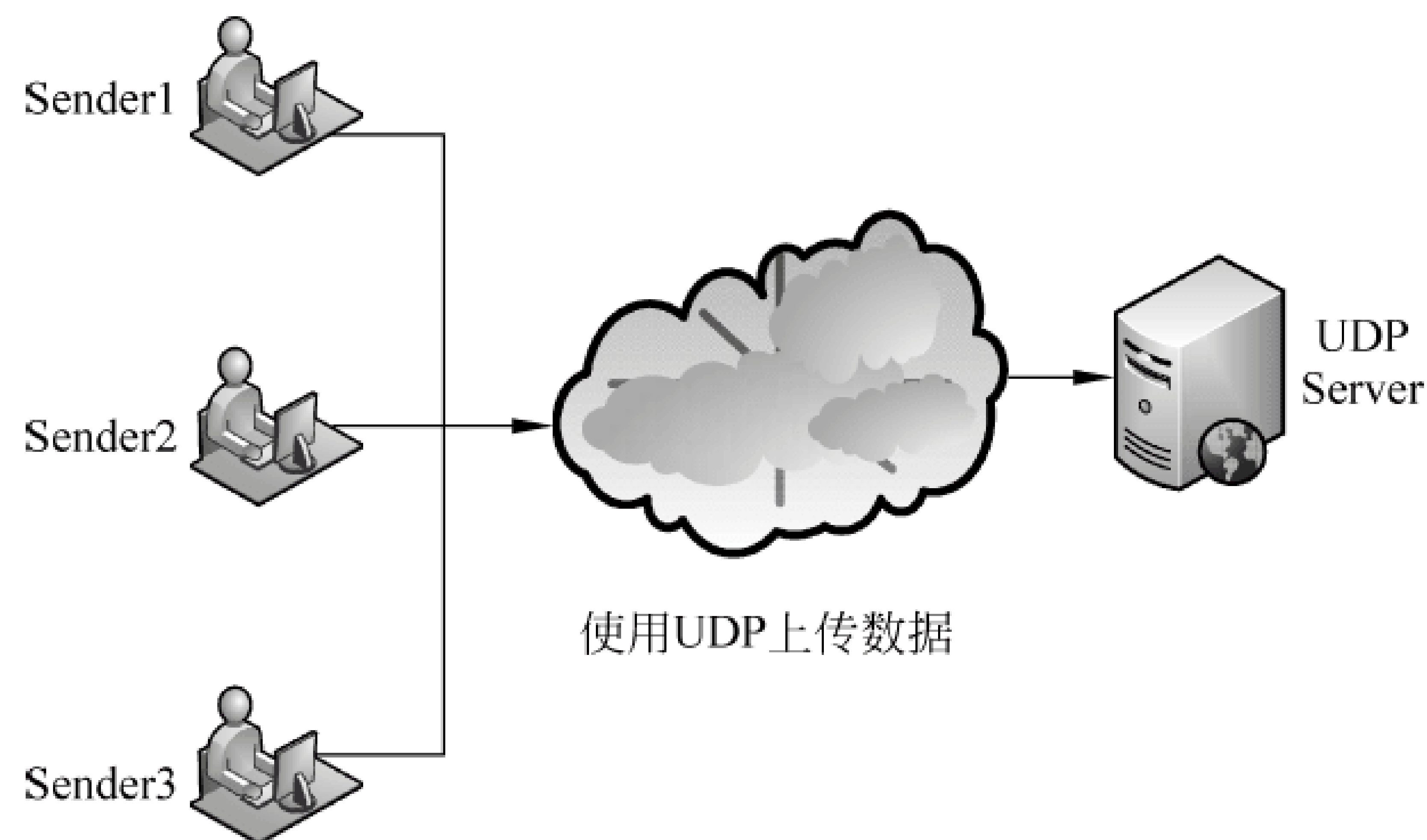


图 9-19 基于 UDP 的接收服务器

1. UDP 接收服务器的代码结构

UDP 接收服务器应当具有如下的代码结构:

```
try { //处理可能出现的勾选异常
      //接收数据前的准备工作:准备好存储数据缓冲区,然后包装成一个数据报包裹对象
      byte buf[ ] = new byte[缓冲区大小];                    //创建一个数据缓冲区
      DatagramPacket pack = new DatagramPacket(buf, buf.length);  //将缓冲区包装成数据报包裹
      //创建数据报套接字对象,然后通过该套接字对象监听某个 UDP 端口,接收数据报包裹
      DatagramSocket ds = new DatagramSocket(UDP 端口号);      //指定被监听的 UDP 端口
      while (运行条件或 true) {    //UDP 接收服务器应一直处于运行状态,随时接收并处理数据
          ds.receive(pack);        //接收发送来的数据报包裹
          ......          //处理所接收到的数据报包裹,可通过网络地址区分出不同发送方(采集点)
      }
      ds.close();                  //关闭数据报套接字
}
catch(IOException e) { System.out.println( e.getMessage() ); }
```

UDP 接收服务器应当一直处于运行状态,这样可以随时接收并处理数据。与例 9-9 所示的普通 UDP 接收方程序相比,UDP 接收服务器只是在代码结构上增加了一个重复接收数据报的循环,其他地方没有区别。可以继续在 UDP 接收服务器的代码结构上增加多线程技术,这样就能提高数据接收的能力。在对 UDP 接收服务器做上述修改后,发送方应用程序的代码结构不需要变动,9.4.1 节例 9-10 的发送方应用程序可以继续使用。

2. UDP 接收服务器的特点

将多个数据采集点的数据上传给数据服务器,可以使用 UDP 上传数据,也可以使用 TCP 上传。和 TCP 相比,使用 UDP 的接收服务器具有如下特点。

- UDP 接收服务器比 TCP 接收服务器运行效率高。使用 TCP 的接收服务器与每个发送方都建立一个 TCP 连接。当有很多个发送方时,服务器需要维护大量的 TCP 连接,这会耗费很多系统资源。
- UDP 接收服务器可能会丢失数据。因网络拥堵等原因,UDP 接收服务器可能会错过发送方上传的数据,即丢失数据。而 TCP 是一种有连接的通信,它能自动重发被丢失的数据,因此 TCP 是一种可靠的数据传输方式。

9.4.3 UDP 多播

使用 UDP 发送数据,通常是一个发送方对一个接收方;或多个发送方对一个接收方(即 UDP 接收服务器)。本节再介绍一种 UDP 独有的应用形式,即一个发送方对多个接收方,这被称为是 **UDP 多播**(或称为 **UDP 组播**)。

1. 多播地址

使用 UDP 多播发送数据,发送方不是将数据发送给某个单一的接收方,而是发向一个**群组**(group)。TCP/IP 使用 IP 地址来标识网络上的某台计算机,但是该如何标识网络上的一个群组呢?

TCP/IP 预留了一些特殊的 IP 地址,专门用于标识网络上的不同群组。这些特殊的 IP 地址被称为**多播地址**(multicast address),或称为 **D 类地址**。多播地址的范围是 224.0.0.0～239.255.255.255。

注:IP 地址分为 A、B、C、D、E 共五类,其中 A、B、C 是基本类,D 类为多播地址,E 类为专用地址。如需了解 IP 地址更多的技术细节,请查阅计算机网络相关的书籍或资料。

2. UDP 多播的通信流程

UDP 多播的应用场景如图 9-20 所示。

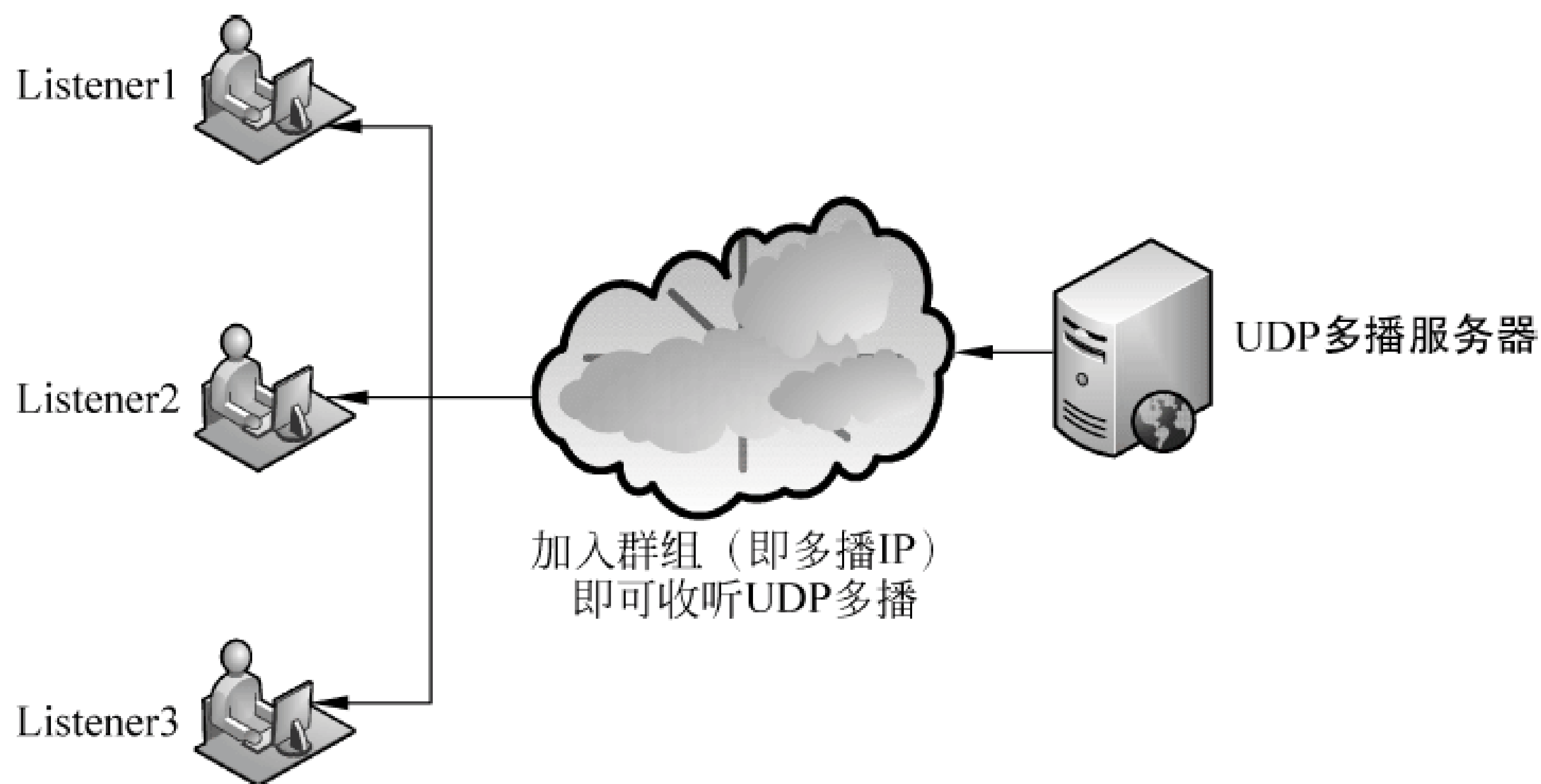


图 9-20　UDP 多播的应用场景

UDP 多播的通信流程如下。

(1) **发送方**应用程序首先确定群组的多播地址和 UDP 端口号,然后使用 UDP 向群组持续发送数据。例如向群组持续发送标准时间,这相当于是用 UDP 实现了一个与 9.3.3 节例 9-7 类似的时间服务器。UDP 多播里的发送方称为**多播服务器**(multicast server)。

(2) **接收方**应用程序首先准备好接收数据用的缓冲区,然后加入发送方指定的**群组**(用多播地址表示),监听该群组指定 UDP 端口的通信,接收群组中发送的数据。这有点类似于在收听某个频道的广播,例如收听时间频道播出的时间报时。UDP 多播里的接收方被称为**收听方**(listener)。

和之前的 UDP 通信程序相比,编写 UDP 多播程序时,发送方(多播服务器)应用程序需要将原来接收方的普通网络地址改成一个表示群组的多播地址;接收方(收听方)应用程序则需要将原来的数据报套接字类 DatagramSocket 换成 Java API 中另一个 UDP 多播专用的套接字类,即多播套接字类 **MulticastSocket**。

3. 多播套接字类 MulticastSocket

收听方应用程序需通过多播套接字对象加入群组,然后才能接收多播群组中发送的数据。请读者阅读下面的多播套接字类 MulticastSocket 说明文档。

java. net. **MulticastSocket** 类说明文档

public class **MulticastSocket**
extends **DatagramSocket**

	修饰符	类成员(节选)	功 能 说 明
1		**MulticastSocket**()	构造方法
2		**MulticastSocket**(int port)	构造方法,指定群组的 UDP 端口
3		void **joinGroup**(InetAddress mcastaddr)	加入多播群组
4		void **receive**(DatagramPacket p)	接收数据报包裹
5		void **send**(DatagramPacket p)	发送数据报包裹
6		void **setTimeToLive**(int ttl)	设置数据报包裹的 TTL(生存)时间
7		void **leaveGroup**(InetAddress mcastaddr)	退出多播群组

…

4. 一个 UDP 多播演示程序

这里给出一个完整的 UDP 多播演示程序,其中包括多播服务器和收听方两个应用程序。为便于调试,程序员可以在同一台计算机主机上运行这两个程序。多播服务器演示程序在运行时会持续向群组(多播地址 224.0.1.1)的 UDP 8000 端口发送标准时间。收听方演示程序启动后会加入这个群组,接收 5 次标准时间。例 9-11 和例 9-12 分别给出了多播服务器和收听方演示程序的 Java 示例代码。

例 9-11 一个 UDP **多播服务器**演示程序的 Java 示例代码(JMulticastServer. java)

```
1  import java.net.*;                                  //导入 java.net 网络包中的类
2  import java.io.*;                                   //导入 java.io 输入输出流包中的类
3  import java.time.*;                                 //导入 java.time 包中与时间相关的类
4  public class JMulticastServer {                     //主类:UDP 多播服务器演示程序
5      public static void main(String[] args) {        //主方法
6          try {                                       //处理可能出现的勾选异常
7              System.out.println("Send multicast data to 224.0.1.1:8000.....\n");
8              //首先准备好多播群组和一个数据报套接字对象
9              InetAddress group = InetAddress.getByName("224.0.1.1");   //生成多播地址
10             DatagramSocket ds = new DatagramSocket();  //创建一个数据报套接字对象
11             //开始多播:播出当前标准时间,总共播出 100 次,每次间隔 1 秒
12             for (int n = 1; n <= 100; n++)  {
13                 LocalDateTime t = LocalDateTime.now(); //假设本机时间为标准时间
14                 String msg = t.getHour() + ":" + t.getMinute() + ":" + t.getSecond();
15                 byte buf[] = msg.getBytes();           //将字符串转换成字节数组
16                 //将标准时间封装成数据报包裹,发送到多播群组的 UDP 端口 8000
17                 DatagramPacket pack = new DatagramPacket(buf, buf.length, group, 8000);
18                 ds.send(pack);                         //将数据报发到多播群组
19                 Thread.sleep(1000);                    //休眠 1 秒
20             }
21             ds.close();                                //关闭数据报套接字
22         } catch(Exception e) { System.out.println( e.getMessage() ); }
23 }  }
```

例 9-12 一个 UDP 多播**收听方**演示程序的 Java 示例代码(JMulticastListener. java)

```
1  import java.net.*;                                  //导入 java.net 网络包中的类
2  import java.io.*;                                   //导入 java.io 输入输出流包中的类
3  public class JMulticastListener {                   //主类:UDP 多播收听方演示程序
4      public static void main(String[] args) {        //主方法
5          try {                                       //处理可能出现的勾选异常
6              System.out.println("Listen multicast data at 224.0.1.1:8000.....\n");
7              //收听前的准备工作:准备好数据缓冲区,然后包装成数据报包裹对象
8              byte buf[] = new byte[128];             //定义接收数据的缓冲区
9              DatagramPacket pack = new DatagramPacket(buf, buf.length);
10             //创建多播套接字对象,然后加入到多播服务器指定的多播群组
11             MulticastSocket ms = new MulticastSocket(8000);  //使用 UDP 8000 端口
12             InetAddress group = InetAddress.getByName("224.0.1.1");  //生成多播地址
13             ms.joinGroup(group);                    //让多播套接字对象 ms 加入多播群组
14             for (int n = 1; n <= 5; n++)  {         //收听 5 次多播
15                 ms.receive(pack);                   //收听多播群组中发送的数据报包裹
16                 //将数据报包裹里的字节数组转成字符串
17                 String msg = new String (pack.getData(), 0, pack.getLength() );
18                 System.out.println(n + " - 收听到的数据:" + msg);
19                 Thread.sleep(1000);                 //休眠 1 秒
20             }
```

```
21                ms.leaveGroup(group);              //退出多播群组
22                ms.close();                        //关闭多播套接字
23                System.out.println("Quit the multicast group at 224.0.1.1:8000.");
24            } catch(Exception e) { System.out.println( e.getMessage() ); }
25    }  }
```

在 Eclipse 集成开发环境中首先运行例 9-11 的多播服务器演示程序,然后再运行例 9-12 的收听方演示程序,收听方将显示从多播群组中接收到的标准时间信息。图 9-21 给出了例 9-12 收听方演示程序的运行结果,其中显示了 5 次从多播群组中接收到的时间数据。

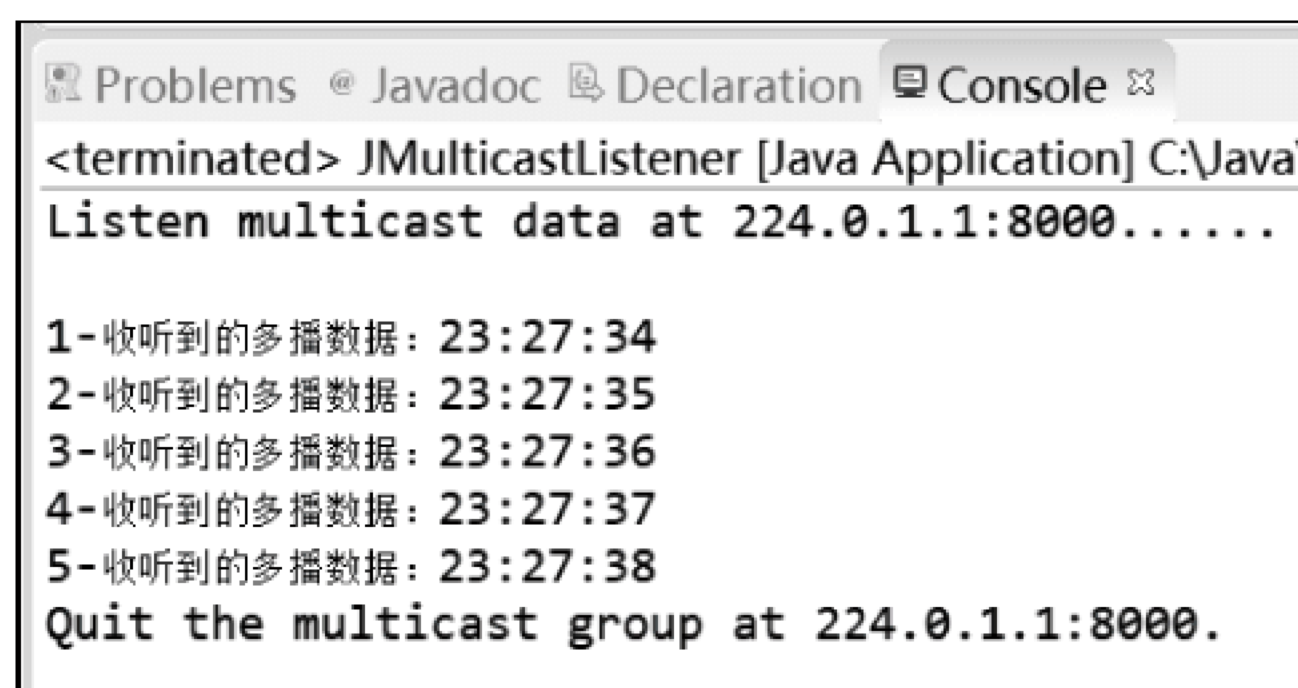


图 9-21　例 9-12 收听方演示程序的运行结果

本节习题

1. 下列关于 TCP 和 UDP 的描述中,错误的是(　　)。
   A. TCP 是有连接的通信　　B. TCP 可以实现双向通信
   C. UDP 是无连接的通信　　D. UDP 不能实现双向通信
2. 数据报包裹类 DatagramPacket 中没有包含的信息是(　　)。
   A. 对方 IP 地址　　B. 对方端口
   C. 被发送的数据　　D. 网络连接状态
3. 数据报包裹类 DatagramPacket 中读取数据的方法是(　　)。
   A. getData()　　B. getInputStream()
   C. accept()　　D. getLength()
4. 数据报套接字类 DatagramSocket 中发送数据的方法是(　　)。
   A. send()　　B. connect()
   C. getOutputStream()　　D. receive()
5. 数据报套接字类 DatagramSocket 中接收数据的方法是(　　)。
   A. send()　　B. getInputStream()
   C. accept()　　D. receive()
6. 下列关于 UDP 通信的描述中,错误的是(　　)。
   A. UDP 通信不需要建立连接
   B. UDP 通信必须先建立连接然后才能通信

C. UDP 通信可以多个发送方对一个接收方

D. UDP 通信可以一个发送方对多个接收方

7. 多播套接字类 MulticastSocket 中加入群组的方法是（　　）。

A. send()　　B. receive()　　C. joinGroup()　　D. leaveGroup()

8. UDP 多播中使用的数据报包裹类是（　　）。

A. DatagramPacket　B. MulticastSocket　C. DatagramSocket　D. Socket

本章学习要点

- 了解计算机网络的基本原理，理解网络编程中常用的概念和术语。
- 学习并掌握基于 TCP 或 UDP 的网络应用程序代码框架，并能熟练运用 Java API 中相关的类进行网络编程。
- 掌握在单台计算机上调试网络应用程序的方法。
- 本章所学习的网络知识已基本能够满足网络编程的需要。如果希望深入学习计算机网络，读者可以进一步选修专门的计算机网络课程。

本章习题

1. 编写程序。模仿 9.1.4 节例 9-1 中的因特网地址类 InetAddress 演示程序，编写一个自己的 Java 程序，用于查询本地主机或因特网上任意一台公共主机的主机名和 IP 地址。

2. 重写程序。阅读并理解 9.2.3 节例 9-3 中的访问 Java 语言网站主页演示程序，然后重写这个程序，查看程序的运行结果。

3. 重写程序。阅读并理解 9.3.3 节例 9-6 和例 9-7 中的 C/S 架构时间服务程序，然后重写这两个程序，查看程序的运行结果。

4. 重写程序。阅读并理解 9.4.1 节例 9-9 和例 9-10 中的 UDP 通信演示程序，然后重写这两个程序，查看程序的运行结果。

5. 重写程序。阅读并理解 9.4.3 节例 9-11 和例 9-12 中的 UDP 多播程序，然后重写这两个程序，查看程序的运行结果。

第10章 数据库编程

数据库应用系统是应用软件开发过程中最为常见的一种系统。图 10-1 给出了一个 C/S 架构数据库应用系统的组成示意图。

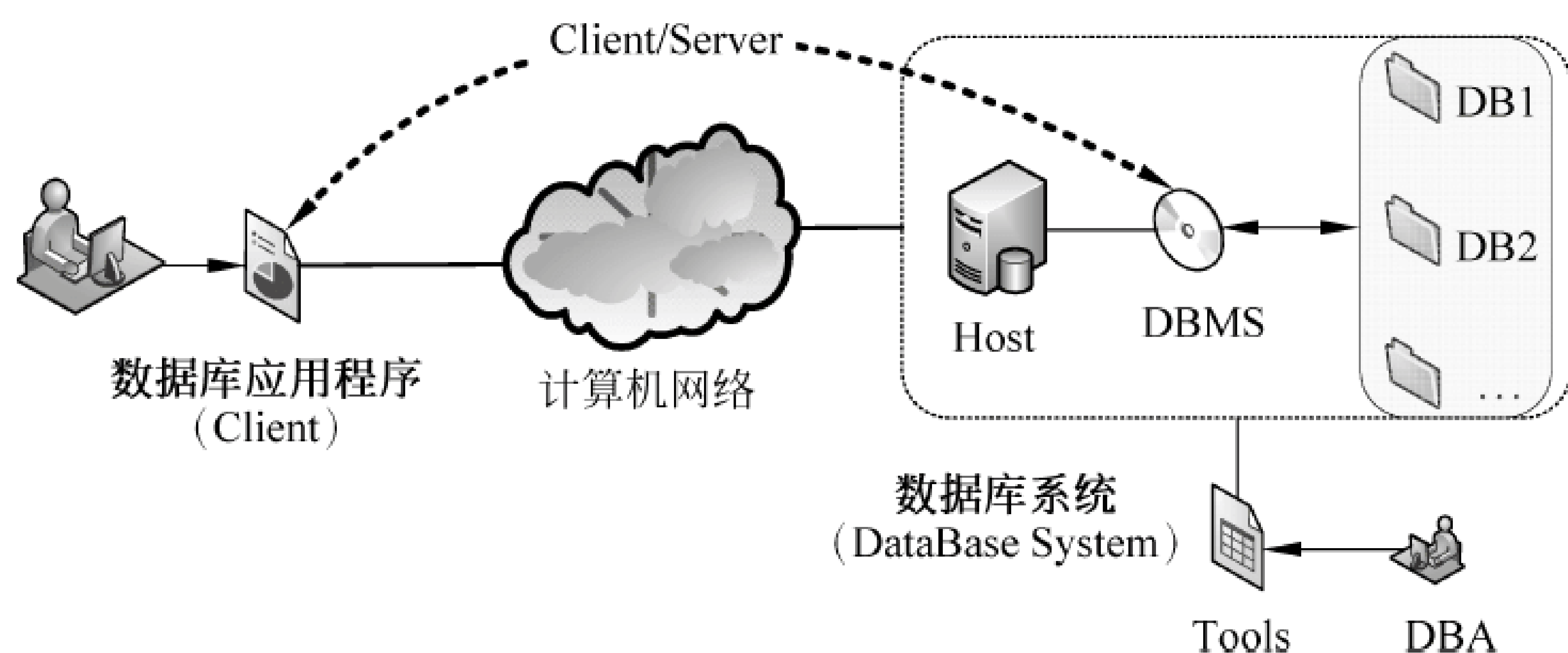


图 10-1　C/S 架构数据库应用系统的组成示意图

数据库应用系统由**数据库系统**和**数据库应用程序**两大部分组成。数据库系统提供数据服务，是服务器端。数据库应用程序使用数据服务，是客户端。

1. 数据库系统

使用计算机，通常是以**文件**形式来存储和管理个人信息的，例如保存在硬盘上的各种 Word 文档、Excel 电子表格或 JPEG 图像文件等。

基于网络的计算机信息系统，例如大学的教务管理系统、企业的电子商务系统等，通常都需要管理大量数据，同时还会有很多人来访问这些数据。**数据库系统**（**D**ata**B**ase **S**ystem，**DBS**）就是一种专门用来存储、管理数据并为他人提供数据服务的系统。

为了科学高效地管理数据，数据库系统首先需要建立**数据模型**，然后依据模型对数据进行集中存储，规范管理。通俗地说，**数据库**就是一种按照某种特定数据模型来组织、存储和管理数据的仓库。目前最常用的数据模型是**关系模型**（relational model），或称为**实体-关系模型**（entity-relationship model）。按照关系模型所建立的数据库被称为**关系型数据库**（relational database）。

2. 数据库应用程序

数据库系统对外提供数据库访问服务。用户访问数据库系统，其目的是操作数据库中的数据。常见的数据库操作有**增**、**查**、**改**、**删**（Create、Read、Update 或 Delete，**CRUD**）。用户

需要通过数据库应用程序才能访问数据库系统，操作数据库中的数据。

本章学习如何编写数据库应用程序，即**数据库编程**。在正式学习数据库编程之前，需要先了解一下数据库系统的基本原理。

10.1　数据库系统的基本原理

数据库系统是计算机专业一门独立的课程，课程内容很多，也很专业。很多读者在学习程序设计之前并没有学过数据库系统课程，不具备学习数据库编程的基础。

针对上述问题，本节以关系型数据库为例，将程序员必须具备的数据库知识提炼出来，以通俗易懂的形式呈现给读者。在掌握了这些数据库知识之后，读者就可以无障碍地学习本章后续数据库编程部分的内容了。

10.1.1　数据库系统的基本组成

数据库系统（参见图 10-1）主要由**主机（Host）**、**数据库管理系统（DMBS）**、**数据库（DB）**、**管理工具（Tools）**等组成。另外，数据库系统还需要配备**数据库管理员（DBA）**，专门负责系统的日常运行维护工作。

1. 主机

数据库系统需要存储、管理大量数据，因此服务器主机首先应当配备大容量存储设备，例如独立冗余磁盘阵列（**R**edundant **A**rray of **I**ndependent **D**isks，**RAID**）。

数据库系统还需要对外提供数据库访问服务，因此服务器主机应当具有一定的网络带宽，并选用性能较高的服务器硬件。

2. 数据库管理系统

数据库管理系统（**D**ata**B**ase **M**anagement **S**ystem，**DBMS**）是一种专门用于定义、操作、存储和管理数据的软件系统，它是数据库系统的核心。DBMS 通常与操作系统、编译系统一起被归为系统软件，而不是应用软件。

数据库系统通过 DBMS 对外提供数据库访问服务，访问数据库系统必须通过 DBMS。从外部看，DBMS 相当于是一个专门提供数据服务的服务器程序。DBMS 一般通过 TCP 对外提供数据库访问服务，并具有相对固定的 TCP 服务端口。

与 DBMS 建立数据库连接的客户端程序有两类：一类是 DBMS 自身提供的管理工具软件；另一类是由程序员开发的数据库应用程序。

- **管理工具软件**。这是 DBMS 为数据库管理员或程序员提供的客户端程序，可以直接与 DBMS 建立连接，访问数据库系统。管理工具软件主要用于设置、维护或测试数据库系统。
- **数据库应用程序**。这是由程序员开发的 DBMS 客户端程序。数据库应用程序必须通过特殊的**应用程序编程接口**（**A**pplication **P**rogramming **I**nterface，**API**）才能与 DBMS 建立连接，访问数据库系统。常用的数据库 API 有 JDBC API 和 ODBC API

两种。数据库应用程序是提供给用户使用的,用户通过数据库应用程序访问数据库中的数据。

3. 数据库

数据库(**D**ata**B**ase,**DB**)是真正存放数据的地方。一个数据库对应磁盘上的一个或多个文件(例如数据文件、控制文件、日志文件等)。

一个数据库管理系统DBMS可以创建并管理多个数据库。例如,中国农业大学有教务管理信息、后勤管理信息和科研项目管理信息等多个不同的数据库,可以使用同一个DBMS对它们进行管理。

4. 管理工具

管理工具(Tools)是数据库管理系统DBMS配套提供的一组工具软件,其功能主要包括创建数据库、查看数据库、数据库备份、账户管理等。数据库管理员使用管理工具软件来设置、维护数据库系统。程序员也可以使用管理工具软件来测试数据库系统,调试程序。

大型计算机信息系统通常都需要使用独立的数据库系统来专门存储和管理数据。数据库系统具有如下特点。

- **数据冗余少**。数据库系统通过集中管理、网络共享、范式设计和重复消除等手段,尽可能地减少数据冗余。
- **数据正确可靠**。数据库系统通过制定完整性约束条件对数据进行检查。只有满足约束条件的数据才能存入数据库,这样可以保证所有存放在数据库中的数据都是正确、可靠的。
- **具有授权访问控制**。数据库系统具有完善的访问控制机制,这样可以保证数据安全,防止信息泄露。只有被授权的用户才能访问数据库系统。通过权限设置,可以控制用户只能访问数据库中的部分数据,即只能访问应当访问的数据。
- **提供数据库访问服务**。数据库系统以网络服务的形式对外提供数据库访问服务。其客户端程序有两类:一类是DBMS自身提供的管理工具软件;另一类是由程序员开发的数据库应用程序。

5. 数据库管理员

数据库管理员(**D**ata**B**ase **A**dministrator,**DBA**)是专门负责数据库系统运行、维护的专业技术人员。与数据库系统相关的其他人员还有:

- **用户**。用户不能直接访问数据库系统,必须通过数据库应用程序才能访问数据库中的数据。用户具有不同的角色和访问权限,例如在大学教务管理系统中,教务人员负责信息的录入、修改和删除,也可以查看大部分教务数据,而教师、学生则只能查看自己的个人信息或课程信息。
- **程序员**。主要负责数据库应用程序开发,即数据库编程。程序员需熟悉数据库系统的基本原理,以及编程所需的程序设计语言(例如Java语言)。不同行业、不同单位的业务类型不同,业务流程不同,因此所使用的数据库应用程序也不同。数据库应用程序通常都是根据用户需求定制开发出来的。目前,计算机行业对程序员的需求

量是最大的。

- **数据库设计人员**。数据库设计一般由经验丰富的资深软件开发人员担任，主要负责数据库系统的需求分析、数据模型设计和系统方案设计等。数据库设计人员首先要懂程序设计，同时还应当具备数据库系统相关的知识。本章所学习的数据库知识已基本能够满足数据库编程的需要。如果想深入学习数据库系统相关的理论和方法，读者可以进一步选修专门的数据库系统课程。

10.1.2　关系型数据库

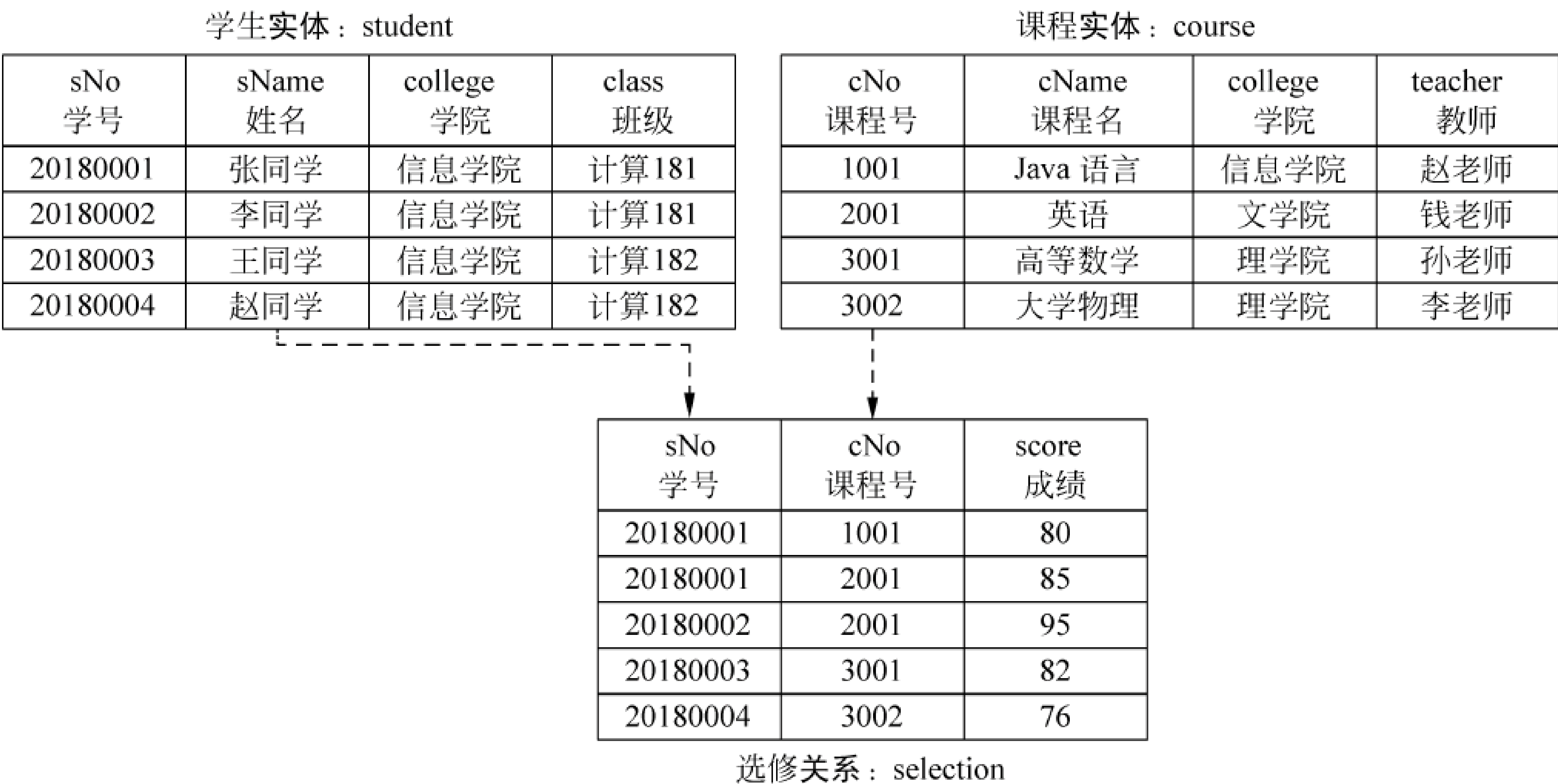
关系型数据库是按照关系模型来描述和存储客观世界中**实体**(entity)以及实体间**关系**(relationship)的数据库系统。例如在大学教务管理系统中，学生是一个实体，课程也是一个实体；学生实体与课程实体之间存在选修关系，即某位学生选修了某门课程。

1. 关系模型

关系模型看起来就像是一个由**行**(row)和**列**(column)组成的二维表格。表格的每一行被称为一条**记录**(record)。一条记录包含多个数据项，每个数据项被称为一个**字段**(field)。同一表格中的记录都具有相同的字段结构。表格中有一个标明字段名称(即列的名称)的表头，称作二维表格的**表结构**(table structure)。

1) 描述和存储实体的表格

可以用表格来描述和存储实体信息。例如，学生是一个实体，可以定义一个学生表 student 来描述和存储学生相关的信息，参见图 10-2。



学生实体：student

sNo 学号	sName 姓名	college 学院	class 班级
20180001	张同学	信息学院	计算181
20180002	李同学	信息学院	计算181
20180003	王同学	信息学院	计算182
20180004	赵同学	信息学院	计算182

课程实体：course

cNo 课程号	cName 课程名	college 学院	teacher 教师
1001	Java 语言	信息学院	赵老师
2001	英语	文学院	钱老师
3001	高等数学	理学院	孙老师
3002	大学物理	理学院	李老师

sNo 学号	cNo 课程号	score 成绩
20180001	1001	80
20180001	2001	85
20180002	2001	95
20180003	3001	82
20180004	3002	76

选修关系：selection

图 10-2　学生、课程及选课信息的关系模型

学生表 student 的表结构描述了学生信息应当包含学号 sNo、姓名 sName、学院 college 和班级 class 等 4 个数据项(即 4 个字段)。表结构后面的每一行都是一条记录，每条记录描述了一名同学的信息。同样，课程也是一个实体，可以用一个课程表 course 来描述和存储

课程相关的信息。

关系模型中的二维表格称为**表**(table),或**数据表**(data table)。数据表中的所有记录都具有相同的字段结构,它们是属于同一实体类型的数据集合。例如,图 10-2 中的学生表 student 是一个学生信息的数据集合,课程表 course 是一个课程信息的数据集合。数据表通过表结构来描述实体类型包含了哪些字段,以及各字段的名称、数据类型和长度等信息。

关系模型要求数据表中不能有重复的记录,即不能有两条完全一样的记录。用于区分不同记录的字段或某几个字段称作数据表的**主键**(**P**rimary **K**ey,PK)。例如,学生表 student 可以将学号 sNo 作为主键。不同学生之间可能会出现重名,但学号必须是唯一的。同理,课程表 course 可以将课程号 cNo 作为主键。主键可以唯一确定数据表中的一条记录。

2) 描述和存储实体间关系的表格

关系模型同样使用二维表格来描述和存储实体之间的关系。例如,学生实体与课程实体之间存在选修关系,即某位学生选修了某门课程。可以使用学号 sNo、课程号 cNo 和成绩 score 来描述学生选修了哪门课程,以及该门课程的成绩,参见图 10-2 中的选修关系表 selection。表中第一条记录的含义是:学号为 20180001 的同学(即学生表中的张同学)选修了课程号为 1001 的课程(即课程表中的 Java 语言),课程成绩为 80 分。选修关系表中的第二条记录则说明该同学还选修了英语(课程号为 2001),成绩为 85 分。

选修关系表 selection 需要通过学号 sNo 和课程号 cNo 两个字段才能区分不同的选课记录,这两个字段合在一起就是选修关系表的主键。另外,学号 sNo、课程号 cNo 分别是学生表 student 和课程表 course 的主键,因此这两个字段也称作**外键**(Foreign Key,FK)。

3) 关系模型的特点

从图 10-2 可以看出,关系模型描述和存储实体以及实体之间的关系,所使用的都是二维表格。关系模型具有如下两大优点。

- **易于理解,易于实现**。关系模型使用人们熟悉的二维表格来描述和存储数据,简单直观。二维表格既便于人理解,也便于计算机实现。
- **具有坚实的理论基础**。关系模型具有一整套基于范式的设计方法和基于集合的关系运算规则。有了这些方法和规则的指导,关系模型的设计工作就有章可循。关系模型是 E. F. Codd 于 1970 年提出的,为此他获得了 1981 年的图灵奖。

2. 关系型数据库

关系型数据库就是按照关系模型建立起来的数据库。一个关系型数据库系统的核心就是其中的关系型数据库管理系统(Rational DBMS,**RDBMS**)软件。为便于理解,将关系型数据库系统与微软的 Excel 电子表格系统做一个类比。

- 关系型数据库管理系统软件 **RDBMS** 类似于电子表格软件 **Excel**。使用 RDBMS 软件可以创建多个数据库,这类似于电子表格软件 Excel 可以创建多个工作簿。所不同的是,RDBMS 通常比 Excel 更庞大,更复杂。
- 一个**数据库**类似于一个 Excel **工作簿**。一个 Excel 工作簿对应硬盘上的一个文件(. xls 或. xlsx),其中可以包含多张工作表。而一个数据库也是对应磁盘上的一个文件(或多个文件),其中可以包含多张数据表。

- 数据库中的一张**数据表**类似于 Excel 工作簿里的一张**工作表**。它们都是由行和列组成的二维表格。所不同的是,数据库中的数据表必须首先定义表结构,指定各字段的名称、数据类型和长度等属性,同时还需要标明哪些字段是主键,然后才能输入数据。RDBMS 对数据表有严格要求,而 Excel 对工作表则没有特别要求。
- 用户可以使用 Excel 软件直接操作电子表格。但是在数据库系统中,用户只能通过配套的数据库管理工具或应用程序间接操作数据库。

关系型数据库管理系统(RDBMS)为应用程序操作数据库设计了一种专门的计算机语言,这就是结构化查询语言(**S**tructured **Q**uery **L**anguage,**SQL**)。常见的数据库操作有**创建**或**删除**数据库、**创建**或**删除**数据表、向表中**插入**新记录、**查询**数据表中的记录、**修改**或**删除**记录等。

3. 常用的关系型数据库管理系统

目前,关系型数据库是主流。常用的关系型数据库管理系统如下。

- **Oracle**,美国甲骨文(Oracle)公司开发,默认服务端口是 TCP 1521。
- **SQL Server**,美国微软(Microsoft)公司开发,默认服务端口是 TCP 1433。
- **MySQL**,是瑞典 MySQL AB 公司(由美国甲骨文公司提供支持)开发的一个开源 DBMS 软件,默认服务端口是 TCP 3306。
- **Access**,是美国微软(Microsoft)公司开发的一种小型数据库管理系统。
- **Java DB**,是 Apache Derby 软件组织开发的一种小型数据库管理系统,目前也是由美国甲骨文公司提供支持。

10.1.3 结构化查询语言

数据库应用程序需要使用结构化查询语言(SQL)来操作关系型数据库。常用的操作有**创建**或**删除**数据库、在数据库中**创建**或**删除**数据表,以及在数据表中**插入**、**修改**、**删除**或**查询**记录等。本节具体介绍这些常用数据库操作的 SQL 语法。**注**:RDBMS 厂家一般还会对标准 SQL 进行不同程度的扩展。

1. 创建或删除数据库

- 创建数据库。

```
CREATE DATABASE  database_name
```

- 删除数据库。

```
DROP DATABASE database_name
```

2. 创建或删除数据表

- 创建数据表。

```
CREATE TABLE  table_name(
    field_name1  data_type1(size),
```

```
    field_name2  data_type2(size),
    field_name3  data_type3(size),
    …
)
```

创建数据表时需定义各字段的名称、数据类型、长度,以及是否主键等。SQL 中有五大类数据类型,它们分别是**字符型**(CHAR、VARCHAR),**数值型**(INT、NUMERIC),**逻辑型**(BIT),**日期型**(DATE、TIME)和**变长型**(BLOB、MEMO)。**注**:不同厂家 RDBMS 所支持的 SQL 数据类型可能会有所不同。

例如,创建图 10-2 中学生表 student 的 SQL 语句如下:

```
CREATE TABLE student(
    sNo CHAR(10) PRIMARY KEY,   sName CHAR(10),
    college VARCHAR(20),   class VARCHAR(20)
)
```

- 删除数据表。

```
DROP TABLE  table_name
```

3. 插入记录

向数据表中插入一条新记录的 SQL 语法如下:

```
INSERT INTO table_name  VALUES(value1, value2, value3,…)
```

例如,向学生表 student 中插入一条新记录的 SQL 语句如下:

```
INSERT INTO student  VALUES('20180001', '张同学', '信电学院', '计算 181')
```

4. 查询记录

- 查询出数据表中的所有记录。

```
SELECT *   FROM table_name
```

- 只查询部分字段。

```
SELECT field_name1, field_name2, …  FROM table_name
```

- 只查询满足条件的记录。

```
SELECT field_name1, field_name2, …  FROM table_name
WHERE condition
```

例如,查询学生表 student 中学号为 20180001 的同学记录的 SQL 语句如下:

```
SELECT *   FROM student
WHERE sNo =  '20180001'
```

5. 修改记录

修改数据表中记录的 SQL 语法如下:

```
UPDATE table_name
SET field_name1 = value1, field_name2 = value2, ...
WHERE condition
```

例如,将学生表 student 中学号为 20180001 的同学的班级修改成"计算 182",其 SQL 语句如下:

```
UPDATE student
SET class = '计算 182'
WHERE sNo = '20180001'
```

6. 删除记录

删除数据表中记录的 SQL 语法如下:

```
DELETE FROM table_name
WHERE condition
```

例如,删除学生表 student 中学号为 20180001 的同学记录的 SQL 语句如下:

```
DELETE FROM student
WHERE sNo = '20180001'
```

本节只是简要介绍了 SQL 中一些最基本的语法。如需详细了解 SQL,请查阅相关的书籍或资料。

本节习题

1. 数据库编程通常指的是编写(　　)。
   A. 数据库管理系统　　B. 数据库管理工具
   C. 数据库应用程序　　D. 数据库测试程序
2. 数据库系统中不包含(　　)。
   A. 数据库管理系统　　B. 数据库管理工具
   C. 数据库应用程序　　D. 数据库管理员
3. 数据库系统中的数据库类似于 Excel 电子表格中的(　　)。
   A. 工作簿　　B. 工作表
   C. 工作表中的一行　　D. 工作表中的一列
4. 数据库系统中的数据表类似于 Excel 电子表格中的(　　)。
   A. 工作簿　　B. 工作表
   C. 工作表中的一行　　D. 工作表中的一列
5. 数据库系统中的记录类似于 Excel 电子表格中的(　　)。
   A. 工作簿　　B. 工作表
   C. 工作表中的一行　　D. 工作表中的一列
6. 下列选项中,可以唯一确定数据表中一条记录的是(　　)。
   A. 主键　　B. 外键

C. 任意一个字段　　D. 第一个字段

7. 在数据库中创建数据表的 SQL 语句是(　　)。

A. SELECT　　B. INSERT

C. UPDATE　　D. CREATE TABLE

8. 查询数据表中记录的 SQL 语句是(　　)。

A. SELECT　　B. INSERT

C. UPDATE　　D. CREATE TABLE

9. 修改数据表中记录的 SQL 语句是(　　)。

A. SELECT　　B. INSERT

C. UPDATE　　D. CREATE TABLE

10. SQL 语句中描述条件的子句是(　　)。

A. WHERE　　B. FROM　　C. VALUES　　D. SET

10.2 JDBC 数据库编程代码框架

用户不能直接访问数据库系统,必须通过数据库应用程序才能访问数据库中的数据。程序员在编写数据库应用程序时需要考虑以下几个问题。

- 如何访问不同 RDBMS 管理的数据库,需要为每一种 RDBMS 编写一个数据库应用程序吗?
- 一个 RDBMS 可以管理多个数据库,数据库应用程序如何指定访问哪个数据库?
- 如何操作数据库中的数据? 数据库应用程序通过 SQL 语句操作数据库中的数据,该如何将 SQL 语句提交给 RDBMS 执行,又该如何取得 RDBMS 返回的结果呢?

为了帮助程序员解决上述问题,Java 语言专门设计了一种关系型数据库连接规范,即 **JDBC**(**J**ava **D**ata**B**ase **C**onnectivity)规范。Java API 配套提供了一组基于 JDBC 规范的数据库类和接口,它们被统称为 JDBC **API**。本节具体讲解 JDBC 规范、JDBC API 以及数据库应用程序的代码框架。

10.2.1 Java 数据库连接规范 JDBC

数据库系统是以网络服务的形式对外提供数据库访问服务的。关系型数据库只定义了操作数据库的 SQL,但并没有为数据库访问服务定义统一的应用层协议。例如,数据库访问的流程是什么,该如何将 SQL 语句提交给 RDBMS 执行,又该如何取得 RDBMS 返回的结果等。JDBC 规范就是专门为数据库访问服务定义的一种统一的应用层协议。

对于关系型数据库管理系统 RDBMS 来说,JDBC 就是一种对外提供数据库访问服务的**“请求-响应”**规范。虽然不同厂家 RDBMS 软件之间的差别很大,但它们对外服务的接口是一样的。每个 RDBMS 都需要提供一个专门的 JDBC **驱动程序**(driver),用于向数据库应用程序提供 JDBC 服务接口。目前市场上常用的 RDBMS 都支持 JDBC 规范,都提供各自的 JDBC 驱动程序。

对程序员来说,JDBC 就是一组通用的数据库编程接口 API,可以访问不同厂家的

RDBMS。Java 语言通过**java.sql**包中的 **DriverManager** 类和 **Connection**、**Statement**、**ResultSet** 等接口实现了这组 API，它们被统称为 **JDBC API**。使用 JDBC API 编写数据库应用程序，只要加载不同的 JDBC 驱动程序，就可以访问不同厂家的数据库系统。即使是开发中途更换 RDBMS 厂家，只需要安装并加载新的 JDBC 驱动程序就可以了。数据库应用程序中操作数据库的程序代码可以保持不变，不需要做任何修改。图 10-3 给出了 JDBC 数据库连接规范的编程模型。

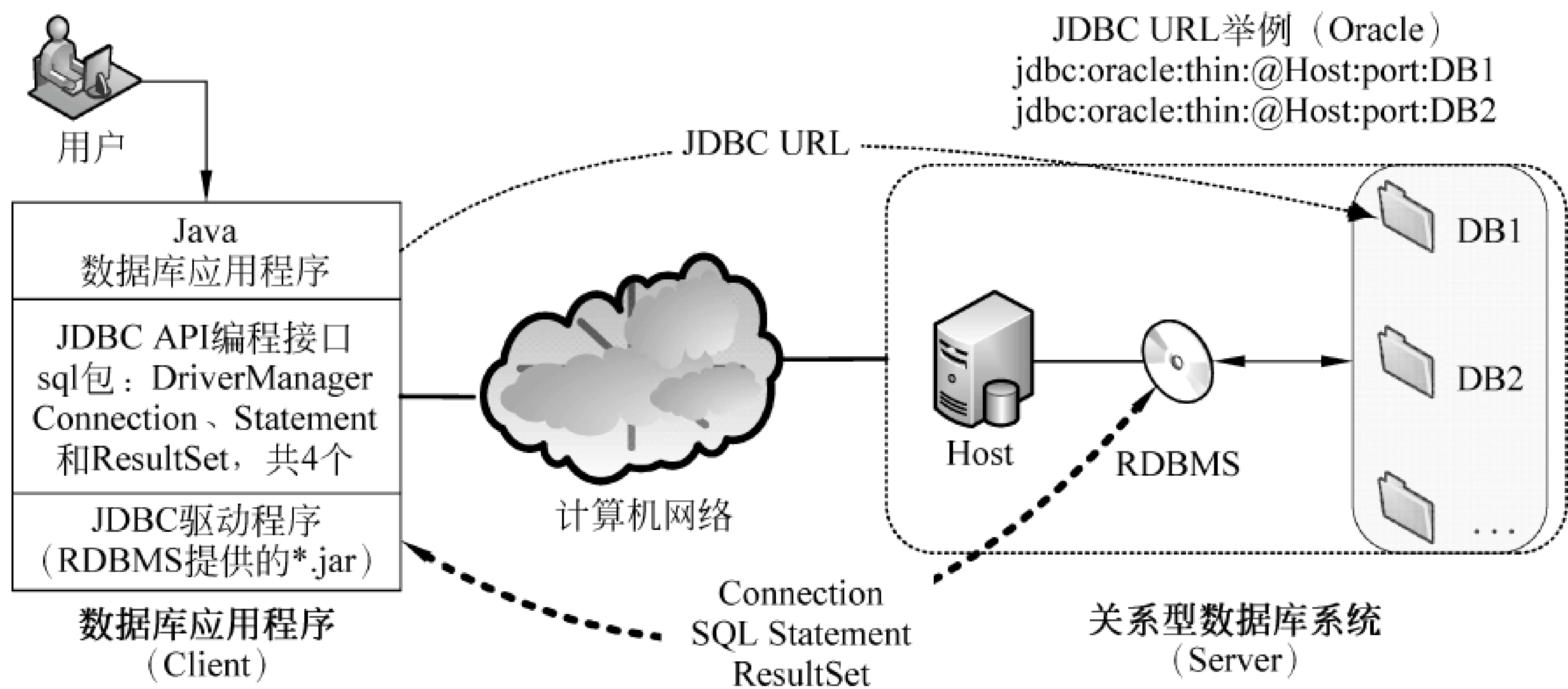


图 10-3　JDBC 数据库连接规范的编程模型

图 10-3 所示 JDBC 数据库连接规范编程模型的说明如下。

- **用户**。用户通过 Java 数据库应用程序访问数据库系统。
- **Java 数据库应用程序**。Java 数据库应用程序调用 JDBC API 编程接口，向 JDBC 驱动程序发送数据库连接和访问请求。程序员以 **JDBC URL** 的形式来指定与数据库系统中的哪个数据库建立连接。所有对数据库的访问请求都是以 SQL 语句(**statement**)形式编写，并通过 JDBC 驱动程序提交给 RDBMS 执行的。
- **JDBC 驱动程序**。JDBC 驱动程序负责与 RDBMS 进行通信，建立数据库连接(**connection**)，将访问数据库的 SQL 语句提交给 RDBMS 执行，然后再将执行结果(例如 **ResultSet**)返回给 Java 数据库应用程序。
- **RDBMS**。关系型数据库管理系统 RDBMS 负责执行 SQL 语句，具体操作数据库(例如 DB1、DB2)中的数据。

10.2.2　JDBC API 编程步骤

使用 JDBC API 编写数据库应用程序主要分 4 步：加载 JDBC 驱动程序、连接数据库、提交 SQL 语句并接收返回结果，最后关闭数据库连接。

1. 加载 JDBC 驱动程序

JDBC 驱动程序实际上是一个 Java 类，加载 JDBC 驱动程序就是加载这个 Java 类的字节码文件(.class 文件)。Java 虚拟机为每个加载到内存的 Java 类创建一个类 **Class** 的对象，该对象保存了 Java 类的全部信息。类 Class 是 Java API 中一个比较特殊的类，它有很

多不同的用途(例如 Java 反射机制)。这里给出类 Class 的说明文档，以便读者对这个类有更多的了解。

java.lang.**Class**<**T**>类说明文档

public final class **Class**<**T**>
extends **Object**
implements **Serializable**，**GenericDeclaration**，**Type**，**AnnotatedElement**

	修 饰 符	类成员(节选)	功 能 说 明
1	static	Class<?> **forName**(String className)	加载指定类名的 Java 类
2	static	Class<?> **forName**(String name, boolean initialize, ClassLoader loader)	加载指定类名的 Java 类
3		String **getName**()	获取类的类名
4		Constructor<?>[] **getConstructors**()	获取类的构造方法数组
5		Field[] **getFields**()	获取类的字段数组
6		Method[] **getMethods**()	获取类的方法数组
7		Annotation[] **getAnnotations**()	获取类的注解数组
8		T **newInstance**()	新建一个该类的对象

…

加载 JDBC 驱动程序只需要使用类 Class 的静态方法成员 **forName**()，加载时给出驱动程序的类名(即驱动程序的.class 字节码文件名)。JDBC 驱动程序由数据库厂家提供，不同 RDBMS 驱动程序的类名不一样。下面列出 5 个常用 RDBMS 的 JDBC 驱动程序类名。

Oracle：oracle.jdbc.driver.OracleDriver。

MySQL：com.mysql.jdbc.Driver。

SQL Server：com.microsoft.sqlserver.jdbc.SQLServerDriver。

Access：sun.jdbc.odbc.JdbcOdbcDriver。**注**：它是基于 JDBC-ODBC Bridge 桥接转换的驱动程序

Java DB：org.apache.derby.jdbc.EmbeddedDriver。**注**：它是基于本地文件系统的驱动程序。

例如，加载 Oracle 数据库驱动程序的示例代码如下：

```
Class.forName( "oracle.jdbc.driver.OracleDriver" );
```

加载 Java DB 数据库驱动程序的示例代码如下：

```
Class.forName( "org.apache.derby.jdbc.EmbeddedDriver" );
```

注 1：早在 20 世纪 90 年代，美国微软公司就提出了一种开放数据库连接(**O**pen **D**ata**B**ase **C**onnectivity，**ODBC**)规范。ODBC 作为一种通用数据库编程接口已经得到了广泛认可。早期，JDBC 主要是经 JDBC-ODBC Bridge(桥接)转换，然后借助 ODBC 驱动来访问不同数据库系统的。目前，JDBC 日趋成熟，各主流 RDBMS 都已直接支持 JDBC 规范。因此 Java 语言从 JDK 1.8 开始建议程序员不要再使用 JDBC-ODBC 桥接方式，并且也不再提供 JDBC-ODBC Bridge 驱动程序。

注 2：Java 语言从 JDK 1.8 开始随安装包提供了一个小型数据库管理系统 Java DB。Java DB 可以直接基于本地文件系统提供数据库访问服务，不需要再额外搭建数据库服务器。建议初学数据库编程的读者使用 Java DB 进行编程实验。

注 3：从 JDBC 4.0 版开始，新的 JDBC 驱动可以被自动加载。因此在使用新版 JDBC 驱动程序时，程序员不需要再显式调用 Class.forName()来加载驱动程序，它会被自动加载。

2. 连接数据库

使用统一资源定位符(URL)可以指定希望访问的网络资源。在数据库系统中，数据库就是其对外服务的网络资源，它是一种数据资源。连接数据库时，JDBC 通过 **JDBC URL** 来指定与数据库系统里的哪个数据库建立连接。JDBC 要求在访问数据库之前必须先建立连接，访问结束后也需要关闭连接，或称断开连接。

1) 数据库的 JDBC URL

大中型 RDBMS，例如 Oracle、MySQL、SQL Server 等，通常是基于 TCP 对外提供数据库访问服务的，其 JDBC URL 中应包含**主机名**(或 IP **地址**)、TCP **端口号**，以及**数据库名**。而小型 RDBMS，例如 Access、Java DB 等，一般是基于本地文件系统提供数据库访问服务的，其 JDBC URL 中应包含数据库文件所在的存储路径及其文件名(或目录名)。

不同 RDBMS 的 JDBC URL 书写格式不太一样，而且差别还比较大。下面列出 5 个常用 RDBMS 的 JDBC URL 书写格式。

Oracle：jdbc:oracle:thin:@host:port:db_name

MySQL：jdbc:mysql://host:port/db_name

SQL Server：jdbc:microsoft:sqlserver://host:port;DatabaseName = db_name

Access：jdbc:odbc:driver = {Microsoft Access Driver (* .mdb, * .accdb)};DBQ = "path/db_filename"

Java DB：jdbc:derby:path/db_foldername; create = true

注：一个 Access 数据库对应本地硬盘上的一个文件，其扩展名为.mdb 或.accdb。一个 Java DB 数据库对应本地硬盘上的一个目录，其下面还会包含若干个子目录。

2) 驱动管理器类 DriverManager

数据库应用程序如果希望连接数据库系统中的某个数据库，这需要通过驱动管理器类 **DriverManager** 中的静态方法 **getConnection**()来实现。调用该方法将在应用程序和数据库之间建立连接，并返回一个接口 **Connection** 的**连接**对象。该对象保存了应用程序和数据库之间的连接信息，或称为连接的**上下文**(context)。后续程序代码将通过这个连接对象来访问数据库。

假设连接主机 192.168.0.10 上的 Oracle 数据库 db1，其示意代码如下：

```
String dbURL = "jdbc:oracle:thin:@192.168.0.10:1521:db1";      //Oracle 数据库 db1 的 URL
Connection con = DriverManager.getConnection(dbURL);            //建立与数据库的连接
```

再如，连接本地主机 D：盘根目录下的 Java DB 数据库 db1，其示意代码如下：

```
String dbURL = "jdbc:derby:D:\\db1; create = true";              //Java DB 数据库 db1 的 URL
Connection con = DriverManager.getConnection(dbURL);            //建立与数据库的连接
```

请读者阅读下面的驱动管理器类 DriverManager 说明文档。

java. sql. **DriverManager** 类说明文档

public class **DriverManager**
extends **Object**

	修　饰　符	类成员(节选)	功 能 说 明
1	static	Connection **getConnection**(String url)	建立数据库连接(不需要账户)
2	static	Connection **getConnection** (String url, String user, String password)	建立数据库连接(需要账户)
3	static	void **setLoginTimeout**(int seconds)	设置等待时间,若超时则连接失败
		…	

3. 提交 SQL 语句并接收返回结果

数据库应用程序在与数据库建立连接之后,可以使用 SQL 语句对数据库进行操作。JDBC 将 SQL 语句分为两大类:一类是**修改**(update)语句,例如创建或删除数据表语句,以及对数据表做插入、修改或删除等操作的 SQL 语句;另一类是**查询**(query)语句,即 SQL 中的 SELECT 语句。

数据库应用程序通过 JDBC API 向 RDBMS 提交 SQL 语句。RDBMS 接收 SQL 语句,并按照 SQL 语句的指令要求执行数据库操作。向 RDBMS 提交 SQL 语句,需要先创建一个接口 **Statement** 的语句对象。调用连接对象的 **createStatement**()方法就可以创建语句对象。例如:

```
Statement s = con.createStatement();  //con是之前已建立好的数据库连接对象
```

数据库应用程序将通过语句对象向 RDBMS 提交 SQL 修改语句或查询语句。

1) 提交 SQL 修改语句

数据库应用程序向 RDBMS 提交 SQL 修改语句的目的是修改数据库,例如创建或删除数据表,对数据表做插入、修改或删除等操作。提交 SQL 修改语句,需要调用语句对象的 **executeUpdate**()方法。例如:

```
s.executeUpdate( "CREATE TABLE student(...)" );       //创建数据表,s为之前创建好的语句对象
s.executeUpdate( "INSERT INTO student VALUES(...)" );//向数据表student中插入一条新记录
```

2) 提交 SQL 查询语句

查询语句就是 SQL 中的 SELECT 语句。数据库应用程序向 RDBMS 提交 SELECT 语句的目的是查询数据库中的数据。RDBMS 执行 SELECT 语句,然后将查询结果以接口 **ResultSet** 的**结果集**对象形式返回给应用程序。提交 SQL 查询语句并接收其查询结果,需要调用语句对象的 **executeQuery**()方法。例如:

```
ResultSet rs = s.executeQuery( "SELECT * FROM student" );//查询数据表student中的所有记录
```

数据库应用程序接收查询结果,对所接收到的结果集对象进行遍历处理,例如遍历显示其内容。对结果集对象进行遍历处理的代码结构如下:

```
while ( rs.next() ) {    //移到结果集的下一条记录.如到末尾则返回 false,循环结束
    ......               //使用 rs.get?()方法读取当前记录中各字段的数据,然后进行处理
}
```

这里还需要说明的是,在语句对象使用结束后,应当关闭语句对象,释放其所占用的资源。例如:

```
s.close();               //关闭语句对象 s
```

4. 关闭数据库连接

数据库应用程序在数据库访问结束后,应当关闭数据库连接,释放其所占用的资源。例如:

```
con.close();             //关闭数据库连接 con
```

本节最后总结一下编写数据库应用程序的 4 个步骤:加载 JDBC 驱动程序、连接数据库、提交 SQL 语句并接收返回结果,最后关闭数据库连接。这里给出其中用到的 3 个 JDBC API 接口说明文档,它们分别是连接接口 **Connection**、语句接口 **Statement** 和结果集接口 **ResultSet**。请读者阅读下面这 3 个重要的 JDBC API 接口说明文档。

java.sql.**Connection** 接口说明文档

public interface **Connection**
extends **Wrapper**, **AutoCloseable**

	修　饰　符	接口成员(节选)	功 能 说 明
1		Statement **createStatement**()	创建 SQL 语句对象
2		PreparedStatement **prepareStatement**(String sql)	创建 SQL 预处理语句对象
3		void **commit**()	提交事务
4		void **rollback**()	回滚事务
5		boolean **isValid**(int timeout)	检查连接是否有效
6		boolean **isClosed**()	检查连接是否已关闭
7		void **close**()	关闭数据库连接

…

java.sql.**Statement** 接口说明文档

public interface **Statement**
extends **Wrapper**, **AutoCloseable**

	修　饰　符	接口成员(节选)	功 能 说 明
1		boolean **execute**(String sql)	将 SQL 语句提交给 DBMS 执行
2		ResultSet **executeQuery**(String sql)	将查询语句提交给 DBMS 执行
3		int **executeUpdate**(String sql)	将修改语句提交给 DBMS 执行
4		void **close**()	关闭语句

…

java. sql. **ResultSet** 接口说明文档			
public interface **ResultSet** extends **Wrapper**，**AutoCloseable**			
	修 饰 符	接口成员(节选)	功 能 说 明
1		int **getRow**()	返回当前记录的行号
2		boolean **first**()	移到结果集的第一行
3		boolean **next**()	移到结果集的下一行
4		boolean **previous**()	移到结果集的上一行
5		boolean **last**()	移到结果集的最后一行
6		boolean **absolute**(int row)	移到指定的行
7		int **getInt**(String columnLabel)	读取字段,返回整数
8		long **getLong**(String columnLabel)	读取字段,返回长整数
9		float **getFloat**(String columnLabel)	读取字段,返回单精度实数
10		double **getDouble**(String columnLabel)	读取字段,返回双精度实数
11		String **getString**(String columnLabel)	读取字段,返回字符串
12		Date **getDate**(String columnLabel)	读取字段,返回日期对象
13		Time **getTime**(String columnLabel)	读取字段,返回时间对象
14		URL **getURL**(String columnLabel)	读取字段,返回 URL 对象
15		void **insertRow**()	将当前行插入数据库
16		void **updateInt**(String columnLabel，int x)	修改字段(整数类型)
17		void **updateDouble**(String columnLabel，double x)	修改字段(实数类型)
18		void **updateString**(String columnLabel，String x)	修改字段(字符串类型)
19		void **updateRow**()	将当前行的数据写入数据库
20		void **deleteRow**()	删除当前行
21		boolean **rowInserted**()	检查当前行是否已被插入
22		boolean **rowUpdated**()	检查当前行是否已被修改
23		boolean **rowDeleted**()	检查当前行是否已经删除
		…	

本节习题

1. 下列关于 JDBC 的描述中,错误的是(　　)。
   A. JDBC 是专门为数据库访问服务定义的一种统一的应用层协议
   B. Java API 配套提供了一组基于 JDBC 规范的类和接口,它们被统称为 JDBC API
   C. 不同 RDBMS 的 JDBC 驱动程序都是一样的
   D. 目前市场上常用的 RDBMS 都支持 JDBC 规范
2. JDBC API 被定义在 Java API 包(　　)当中。
   A. java. sql　　B. java. lang　　C. java. util　　D. java. database
3. 下列选项中,(　　)是 JDBC API 中定义的类。
   A. DriverManager　　B. Connection　　C. Statement　　D. ResultSet
4. 下列选项中,(　　)不是 JDBC API 中定义的接口。

A. DriverManager　B. Connection　C. Statement　D. ResultSet

5. 连接数据库需要用到(　　)中定义的方法。

A. DriverManager　B. Connection　C. Statement　D. ResultSet

6. 创建 JDBC 语句对象需要用到(　　)中定义的方法。

A. DriverManager　B. Connection　C. Statement　D. ResultSet

7. JDBC API 中查询语句 SELECT 返回的结果是(　　)的对象。

A. DriverManager　B. Connection　C. Statement　D. ResultSet

8. 接口 Statement 中将 SQL 查询语句提交给 RDBMS 执行的方法是(　　)。

A. executeQuery()　B. executeUpdate()　C. SELECT　D. UPDATE

10.3　JDBC 数据库编程实验

本节设计了一个 JDBC 编程实验，请读者跟随实验步骤自己动手编写一个完整的 Java 数据库应用程序。编程实验分以下 5 个步骤完成。

(1) 搭建数据库编程实验环境，新建一个数据库编程实验用的 Java 项目。

(2) 为数据库编程实验项目导入所需的外部 JAR 包。

(3) 创建数据库和数据表，向数据表插入记录。

(4) 查询数据表。

(5) 修改、删除记录。

10.3.1　搭建数据库编程实验环境

编写 Java 数据库应用程序需要读者在自己的计算机上建立起 Java 语言开发环境，其中包括 Java 开发包(JDK)和 Java 集成开发环境(IDE)。数据库应用程序还需要连接数据库系统，可以连接网络上某个已经搭建好的数据库系统服务器，也可以在自己的计算机上搭建一个新的数据库系统。本编程实验推荐使用 Java DB 在自己的计算机上搭建一个全新的数据库系统。

请读者按照以下 5 项实验要求搭建好自己的 JDBC 数据库编程实验环境。

(1) **Java 开发包**。下载并安装 JDK 1.8 或更新版本，参见第 1 章 1.2 节。

(2) **Java 集成开发环境**。下载并安装 Eclipse 4.7 或更新版本，参见第 1 章 1.4 节。

(3) **数据库系统**。使用 JDK 1.8 或更新版本自带的 Java DB 关系型数据库管理系统。在安装 JDK 1.8 时采用**默认**安装选项，Java DB 数据库管理系统就会被自动安装在 JDK 1.8 安装目录下的 db 子目录中。

(4) **新建数据库编程实验项目**。在 Eclipse 集成开发环境中新建一个 Java 项目，用于编写数据库应用程序。假设将这个 Java 项目命名为 Chapter10。

(5) **为数据库编程实验项目导入外部 JAR 包**。Java DB 的 JDBC 驱动程序是以 JAR 包文件形式提供的。在 JDK 1.8 安装目录的 db\lib 子目录下有一个名为 derby.jar 的 JAR 包文件，这个文件就是 Java DB 的 JDBC 驱动程序。数据库编程实验项目 Chapter10 需要先导入这个外部 JAR 包文件，然后才能加载其中的驱动程序。

10.3.2 为 Java 项目导入外部 JAR 包

本节讲解如何在 Eclipse 集成开发环境中为 Java 项目导入外部 JAR 包。

除了 JDK 提供的 Java API 之外,编写 Java 应用程序还经常用到各种第三方开发的类库。这些类库通常是以 Java 归档文件(扩展名为.jar,俗称 JAR 包)的形式提供的。Eclipse 集成开发环境将这些第三方类库的 JAR 包称作**外部 JAR 包**(external JARs)。

因为外部 JAR 包可能安装在硬盘的任何目录下,Eclipse 集成开发环境无法确定这些外部 JAR 包文件的存放位置。如果一个 Java 项目需要用到外部 JAR 包,则必须为该项目做一次导入操作,将外部 JAR 包的安装目录添加到 Java 项目的组建路径(Java Build Path 中)。

下面就以 Java DB 驱动程序为例具体演示如何为一个 Java 项目导入外部 JAR 包。导入外部 JAR 包的操作选项被 Eclipse 放在 Java 项目的**属性**(properties)**页**对话框中。

1. 进入 Java 项目的属性页对话框

在 Eclipse 集成开发环境中,首先选中需要导入外部 JAR 包的 Java 项目,然后选择 **Project→Properties**,进入项目的属性页对话框。在这个属性页对话框中可以设置 Java 项目的各种属性。图 10-4 给出了数据库编程实验项目 Chapter10 的属性页对话框。首先在图 10-4 所示属性页对话框的左侧首先选中 **Java Build Path**,然后单击对话框界面右侧的 **Add External JARs** 按钮,进入 **JAR 包选择**对话框。

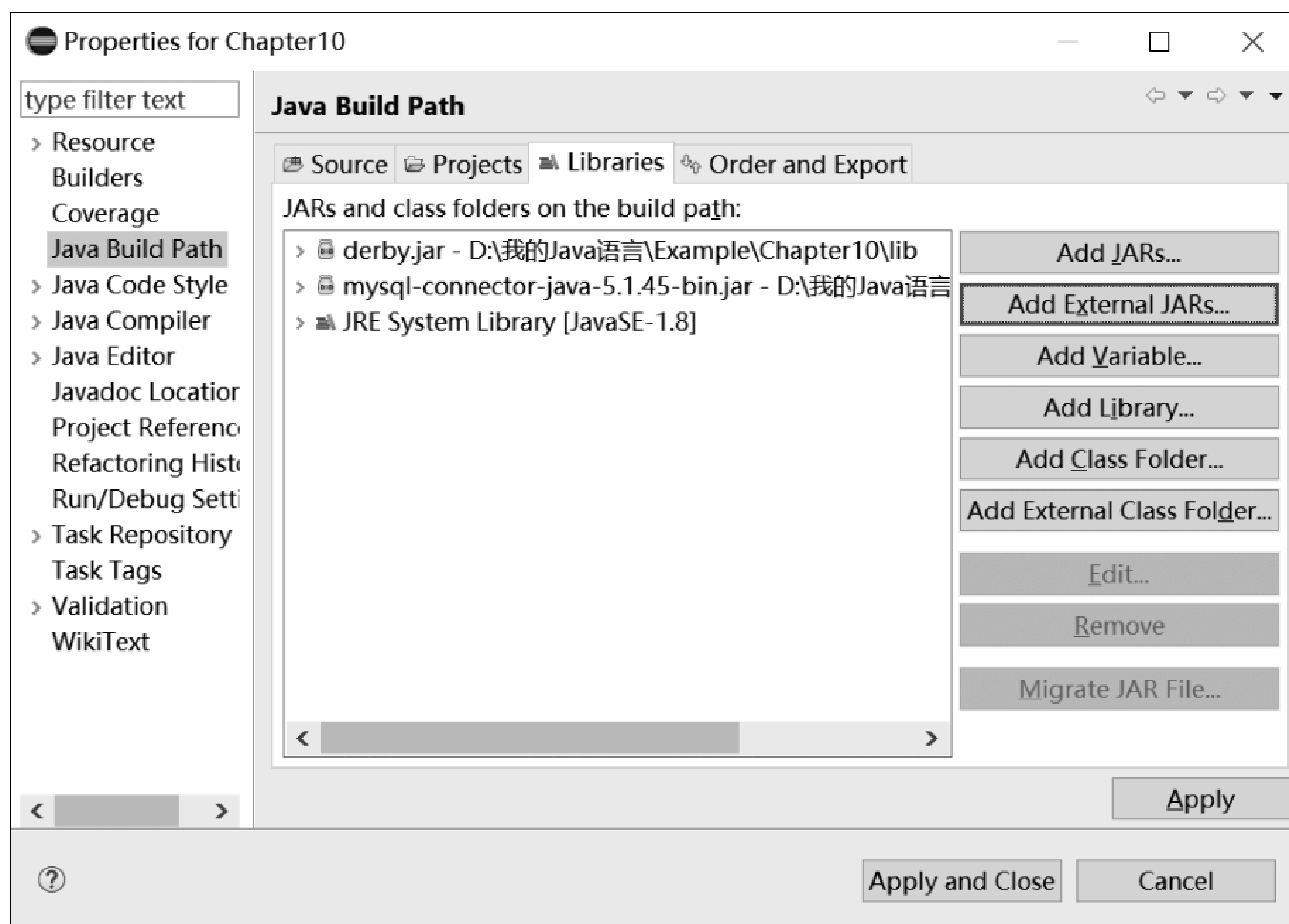


图 10-4 项目 Chapter10 的属性页对话框

2. 选择 JAR 包文件

在对话框中为项目选择需要添加的外部 JAR 包，如图 10-5 所示。在图 10-5 所示的 JAR 包选择对话框中找到 Java DB 驱动程序所在的目录，即 JDK 1.8 安装目录下的 **db\lib** 子目录，选择其中的文件 **derby.jar**，然后单击“**打开**”按钮。

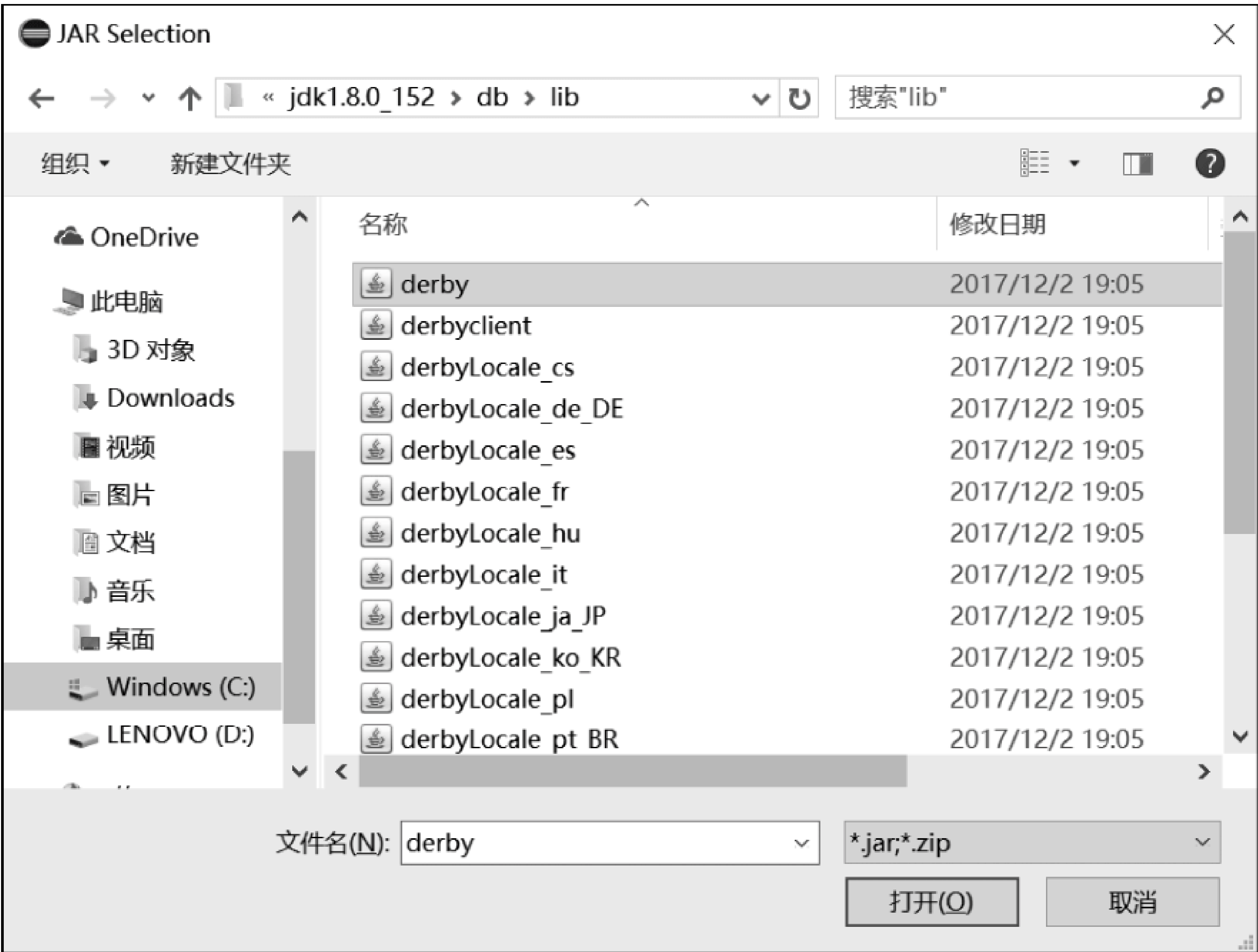


图 10-5 添加外部 JAR 包时的 JAR 包选择对话框

这样就完成了为数据库编程实验项目 Chapter10 导入 Java DB 驱动程序 JAR 包的操作。

3. 查看项目已导入的外部 JAR 包

在 Eclipse 集成开发环境中选中需要查看的 Java 项目，然后选择项目下的 **Referenced Libraries**，其中将显示所有已导入的外部 JAR 包。图 10-6 显示了项目 Chapter10 中已导入的外部 JAR 包文件。

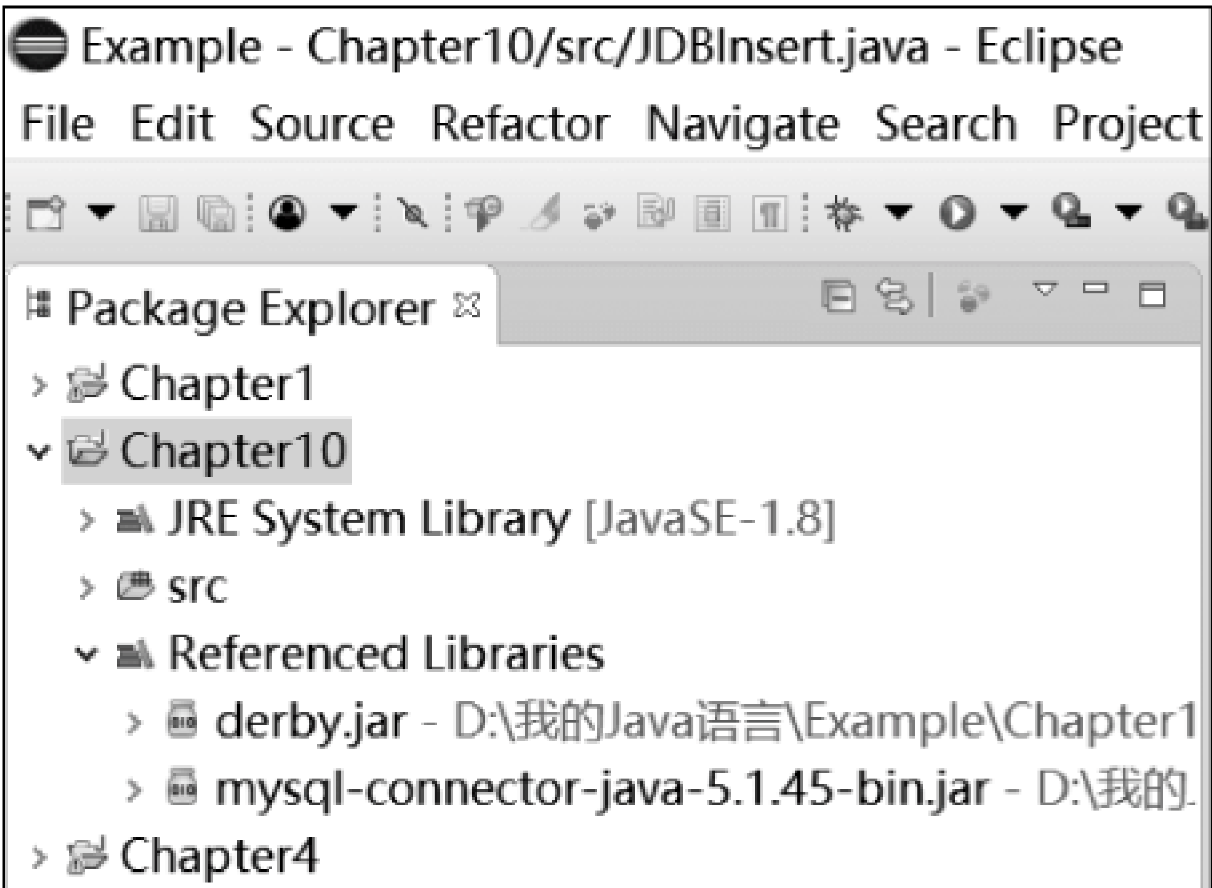


图 10-6 项目 Chapter10 中已导入的外部 JAR 包文件

图 10-6 的显示结果表明,数据库编程实验项目 Chapter10 已导入了两个外部 JAR 包文件。其中的 **derby.jar** 就是刚刚导入的 Java DB 驱动程序 JAR 包。为了对比,作者在此之前还导入了另一个外部 JAR 包 **mysql-connector-java-5.1.45-bin.jar**,这是 MySQL 数据库的 JDBC 驱动程序 JAR 包。

10.3.3 创建数据库和数据表

本节的实验目的是模拟数据库应用系统中的数据库初始创建环节。具体实验内容如下。

(1) **创建数据库**。在本地 Java DB 数据库系统中创建一个名为 cau 的教务信息数据库。

(2) **创建数据表**。在教务信息数据库 cau 中创建一个保存学生信息的数据表 student,其关系模型请参见 10.1.2 节的图 10-2。学生表 student 的表结构包含学号 sNo、姓名 sName、学院 college 和班级 class,共 4 个字段。

(3) **插入新记录**。在学生表 student 中插入 4 条新记录,分别保存 4 名同学的信息。这 4 名同学的具体信息请参见 10.1.2 节的图 10-2。

例 10-1 给出一个完成上述实验内容的 Java 示例程序。

例 10-1 在本地 Java DB 数据库系统中创建教务信息数据库的 Java 示例程序(JDBCreate.java)

```
1   import java.sql.*;                               //导入 java.sql 包中数据库相关的类和接口
2   public class JDBCreate {                         //主类:初始创建一个教务信息数据库 cau
3       public static void main(String[] args) {     //主方法
4           try {  //处理数据库访问过程中可能出现的勾选异常
5               //指定 Java DB 驱动程序的类名,然后调用 Class.forName()加载驱动程序
6               String dbDriver = "org.apache.derby.jdbc.EmbeddedDriver";
                                                     //Java DB 驱动
7               Class.forName(dbDriver);             //加载驱动
8               System.out.println(dbDriver + " loaded.");
9               //连接 Java DB 数据库 cau(对应文件夹 D:\\cau).如不存在则创建
10              String dbURL = "jdbc:derby: D:\\cau; create = true";
                                                     //数据库 cau 的 JDBC URL
11              Connection con = DriverManager.getConnection(dbURL);
                                                     //建立数据库连接
12              System.out.println(dbURL + " connected.");
13              //创建学生表,包含学号、姓名、学院和班级,共 4 个字段
14              Statement s = con.createStatement();      //首先创建语句对象
15              String sqlCreateTable = " CREATE TABLE student( " +
16                      sNo CHAR(10) PRIMARY KEY, sName CHAR(10)," +
17                      college VARCHAR(20), class VARCHAR(20) )"; //SQL:创建新数据表
18              s.executeUpdate(sqlCreateTable);      //向 Java DB 提交 SQL 修改语句
19              System.out.println("TABLE student created.");
20              //下面开始向学生表 student 插入 4 条新记录,分别保存 4 名同学的信息
21              //插入、保存第 1 名同学信息的记录
22              String sqlInsert = "INSERT INTO student VALUES( " +   //SQL:插入新记录
```

```
23                  '20180001', '张同学', '信息学院', '计算 181');
24              s.executeUpdate(sqlInsert);           //向 Java DB 提交 SQL 修改语句
25              //插入、保存第 2 名同学信息的记录
26              sqlInsert = "INSERT INTO student VALUES( " +
27                  '20180002', '李同学', '信息学院', '计算 181');
28              s.executeUpdate(sqlInsert);
29              //插入、保存第 3 名同学信息的记录
30              sqlInsert = "INSERT INTO student VALUES( " +
31                  '20180003', '王同学', '信息学院', '计算 182');
32              s.executeUpdate(sqlInsert);
33              //插入、保存第 4 名同学信息的记录
34              sqlInsert = "INSERT INTO student VALUES( " +
35                  '20180004', '赵同学', '信息学院', '计算 182');
36              s.executeUpdate(sqlInsert);
37              System.out.println("4 student records inserted.");
38              //数据库访问结束,关闭 SQL 语句对象和数据库连接对象
39              s.close();   con.close();
40          }
41          catch(ClassNotFoundException e) {   e.printStackTrace();   } //驱动程序加载异常
42          catch(SQLException e) {   e.printStackTrace();   }   //SQL 语句执行异常
43  }  }
```

请读者阅读并理解例 10-1 中的 Java 程序代码,然后在 Eclipse 集成开发环境中重写并运行这个程序,检查数据库的创建结果。

注:例 10-1 所示的 Java 程序必须被放在数据库编程实验项目 Chapter10 中才能正常运行,因为它要用到之前导入的 Java DB 驱动程序 JAR 包。在重写例 10-1 的 JDBCreate.java 程序时,读者需要在数据库编程实验项目 Chapter10 中新建 Java 类 JDBCreate,然后再输入程序代码。同理,下面的例 10-2 和例 10-3 也需要放在数据库编程实验项目 Chapter10 中才能正常运行。

例 10-1 在 D:盘根目录下创建一个名为 cau 的 Java DB 教务信息数据库,然后在这个数据库中创建学生表 student 并插入 4 条保存学生信息的记录。Java DB 可以直接基于本地文件系统提供数据库访问服务。一个 Java DB 数据库对应本地文件系统的一个目录,其目录名就是数据库名。例如,教务信息数据库 cau 所对应的目录名就是 cau,即“D:\cau”,如图 10-7 所示。

从外部可以看到 Java DB 数据库目录下还包含一些文件和子目录,但看不到数据库内部更多的存储细节,例如数据表 student 是如何存储的。请读者对照图 10-7 检查自己程序所创建的数据库是否正确。

10.3.4　查询数据表

本节的实验目的是模拟数据库应用系统中用户查询数据库的环节。具体实验内容如下。

(1) **查询全部记录**。查询并显示出例 10-1 所创建学生表 student 中的全部学生记录。

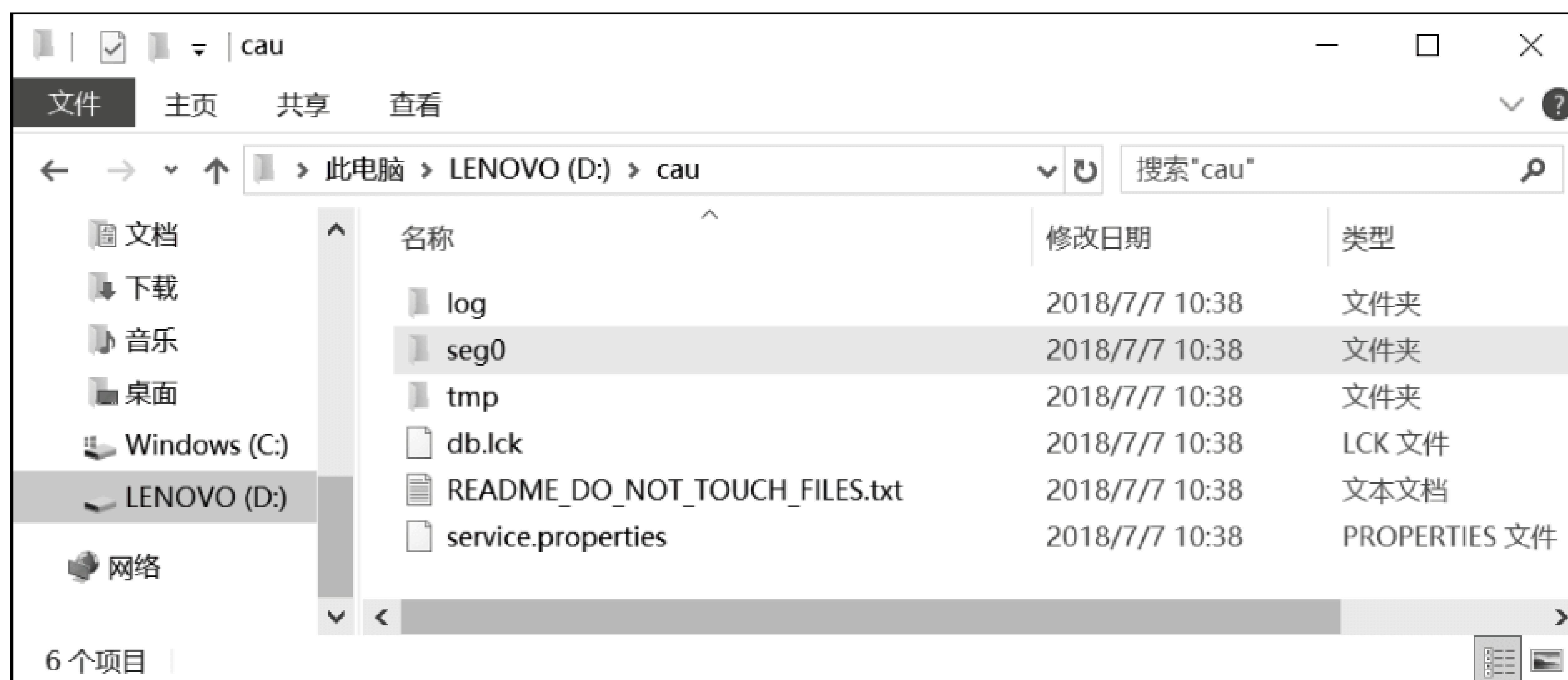


图 10-7 教务信息数据库 cau 所对应的目录内容(Java DB 数据库系统)

(2) **查询满足特定条件的记录**。查询并显示出例 10-1 所创建学生表 student 中"计算181"班同学的学号和姓名。

例 10-2 给出一个完成上述查询功能的 Java 示例程序。

例 10-2 查询并显示学生表 student 中学生记录的 Java 示例程序(JDBSelect.java)

```
1   import java.sql.*;                    //导入 java.sql 包中数据库相关的类和接口
2   public class JDBSelect {              //主类:查询并显示学生表 student 中的学生记录
3       public static void main(String[] args) {          //主方法
4           try {  //处理数据库访问过程中可能出现的勾选异常
5               String dbDriver = "org.apache.derby.jdbc.EmbeddedDriver"; //Java DB 驱动
6               Class.forName(dbDriver);                  //加载驱动
7               System.out.println(dbDriver + " loaded.");
8               String dbURL = "jdbc:derby:D:\\cau; create=true"; //数据库 cau 的 JDBC URL
9               Connection con = DriverManager.getConnection(dbURL); //连接数据库
10              System.out.println(dbURL + " connected.");
11              //查询并显示出学生表 student 中的全部学生记录
12              Statement s = con.createStatement();       //创建语句对象
13              String sqlSelect = "SELECT * FROM student";//SQL 语句:查询全部记录
14              ResultSet rs = s.executeQuery(sqlSelect); //向 Java DB 提交 SQL 查询语句
15              //查询语句会返回查询到的结果集 ResultSet,遍历并显示其中的记录
16              System.out.println("学生表 student 中的全部记录如下:");
17              while ( rs.next() ) {      //移到下一记录,如到末尾则返回 false,循环结束
18                  System.out.print( rs.getString("sNo") +"\t" );    //字段:学号 sNO
19                  System.out.print( rs.getString("sName") +"\t" ); //字段:姓名 sName
20                  System.out.print( rs.getString("college") +"\t" );//字段:学院 college
21                  System.out.println( rs.getString("class") );     //字段:班级 class
22              }
23              //查询并显示出学生表 student 中"计算 181"班同学的学号和姓名
```

```
24              sqlSelect = "SELECTsNo, sName FROM student " + //SQL：查询学号和姓名
25                          WHERE class = '计算 181'";      //查询条件：班级 = '计算 181'
26              rs = s.executeQuery(sqlSelect);             //向 Java DB 提交 SQL 查询语句
27              System.out.println("学生表 student 中"计算 181"班同学的学号和姓名：");
28              while ( rs.next() ) {                       //遍历并显示结果集 rs
29                  System.out.print( rs.getString("sNo") +"\t" ); //字段：学号 sNO
30                  System.out.println( rs.getString("sName") );     //字段：姓名 sName
31              }
32              //数据库访问结束，关闭 SQL 语句对象和数据库连接对象
33              s.close();  con.close();
34          }
35          catch(ClassNotFoundException e) {  e.printStackTrace();  } //驱动程序加载异常
36          catch(SQLException e) {  e.printStackTrace();  }        //SQL 语句执行异常
37  }  }
```

在 Eclipse 集成开发环境中运行例 10-2 的数据库查询程序，查询结果如图 10-8 所示。请读者阅读并理解例 10-2 中的 Java 程序代码，然后在 Eclipse 集成开发环境中重写这个程序，对照图 10-8 检查自己程序的运行结果。

```
Problems  Javadoc  Declaration  Console
<terminated> JDBSelect [Java Application] C:\Java\jre1.8
org.apache.derby.jdbc.EmbeddedDriver loaded.
jdbc:derby:D:\cau; create=true connected.
学生表student中的全部记录如下：
20180001        张同学    信息学院    计算181
20180002        李同学    信息学院    计算181
20180003        王同学    信息学院    计算182
20180004        赵同学    信息学院    计算182
学生表student中"计算181"班同学的学号和姓名：
20180001        张同学
20180002        李同学
```

图 10-8　运行例 10-2 数据库查询程序所得到的查询结果

10.3.5　修改或删除记录

本节的实验目的是模拟数据库应用系统中用户修改或删除数据库数据的环节。具体实验内容如下。

(1) **修改记录**。将例 10-1 所创建学生表 student 中学号为 20180001 的同学的姓名由“张同学”改为“章同学”。

(2) **删除记录**。删除例 10-1 所创建学生表 student 中学号为 20180002 的同学的记录。

例 10-3 给出一个完成上述实验内容的 Java 示例程序。

例 10-3　修改和删除学生表中学生记录的 Java 示例程序(JDBUpdate.java)

```
1   import java.sql.*;                           //导入 java.sql 包中数据库相关的类和接口
2   public class JDBUpdate {                     //主类：修改或删除学生表中的学生记录
3       public static void main(String[] args) { //主方法
```

```
4         try {  //处理数据库访问过程中可能出现的勾选异常
5             String dbDriver = "org.apache.derby.jdbc.EmbeddedDriver";    //Java DB 驱动
6             Class.forName(dbDriver);                         //加载驱动
7             System.out.println(dbDriver + " loaded.");
8             String dbURL = "jdbc:derby:D:\\cau; create = true";  //数据库 cau 的 JDBC URL
9             Connection con = DriverManager.getConnection(dbURL);  //连接数据库
10            System.out.println(dbURL + " connected.");
11            //修改：将学号为 20180001 的同学的姓名由"张同学"改为"章同学"
12            Statement s = con.createStatement();        //创建语句对象
13            String sqlUpdate = "UPDATE student " +      //SQL 修改语句：修改记录
14                          SET sName = '章同学' WHERE sNo = '20180001'";
15            s.executeUpdate(sqlUpdate);                 //向 Java DB 提交 SQL 修改语句
16            System.out.println("A student record updated.");
17            //删除记录：删除学生表中学号为 20180002 的同学的记录
18            String sqlDelete = "DELETE FROM student " +  //SQL 修改语句：删除记录
19                          WHERE sNo = '20180002'";
20            s.executeUpdate(sqlDelete);                 //向 Java DB 提交 SQL 修改语句
21            System.out.println("A student record deleted.");
22            //数据库访问结束,关闭 SQL 语句对象和数据库连接对象
23            s.close();  con.close();
24        }
25        catch(ClassNotFoundException e) {  e.printStackTrace();  }  //驱动程序加载异常
26        catch(SQLException e) {  e.printStackTrace();  } //SQL 语句执行异常
27  } }
```

在 Eclipse 集成开发环境中运行例 10-3 的数据库修改和删除程序，然后再次运行例 10-2 的数据库查询程序，查询结果如图 10-9 所示。请读者阅读并理解例 10-3 中的 Java 程序代码，然后在 Eclipse 集成开发环境中重写这个程序，对照图 10-9 检查自己程序的运行结果。

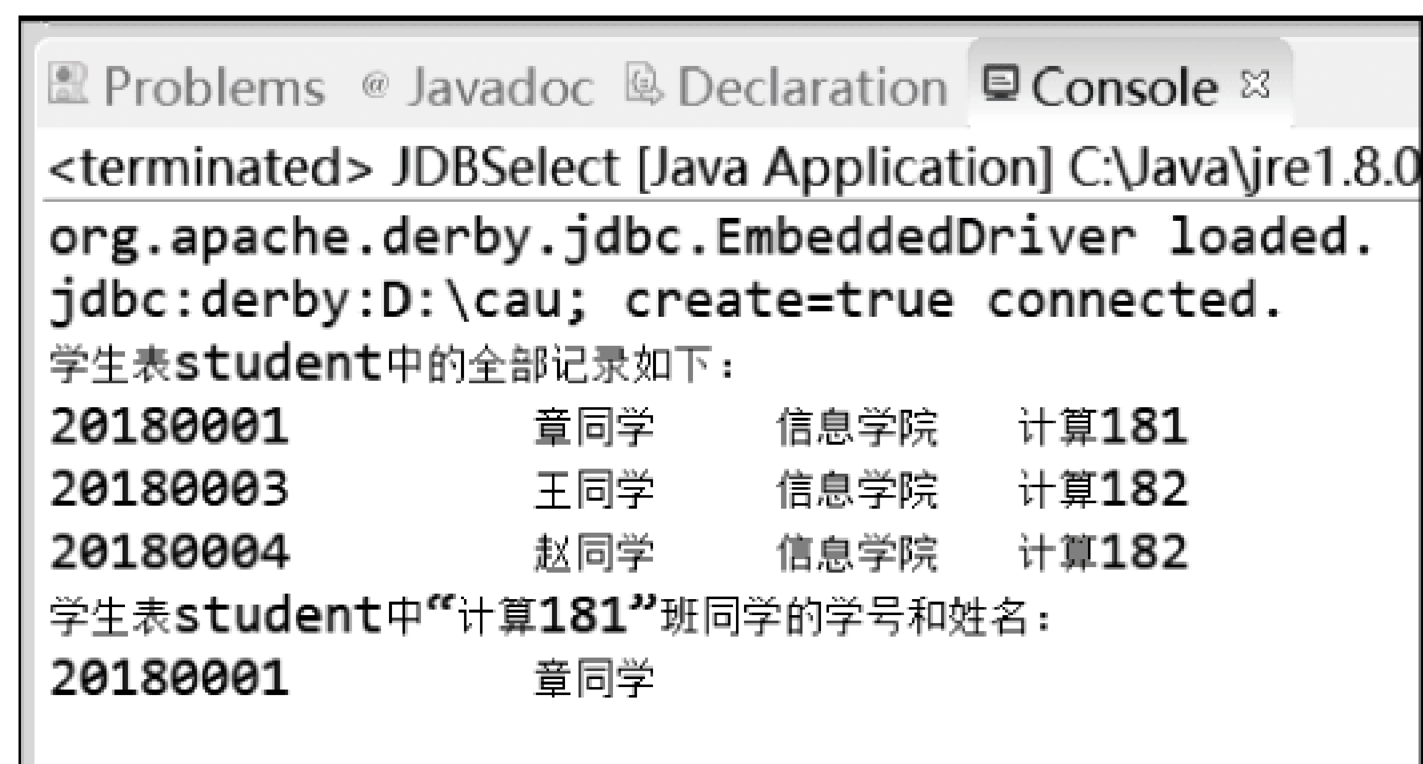


图 10-9 再次查询被例 10-3 修改后数据库的查询结果

为便于用户操作，数据库应用程序通常都设计成图形用户界面。例如，可以将例 10-2 的数据库查询程序修改成图形用户界面，然后使用 swing 图形组件中的 JTable 来显示学生记录清单。关于图形用户界面程序，请参见第 6 章。

本节习题

1. 下列选项中,(   )不是搭建 JDBC 数据库编程实验环境必需的工作。
   A. 安装 JDK　　B. 安装 JDBC 驱动
   C. 导入 JDBC 驱动 JAR 包　　D. 连接网络
2. 下列 Java 包中,Eclipse 将(   )称为外部 JAR 包。
   A. java.sql　　B. java.lang
   C. java.util　　D. org.apache.derby.jdbc
3. 随 JDK 1.8 提供的数据库管理系统是(   )。
   A. Oracle　　B. SQL Server　　C. MySQL　　D. Java DB
4. Java DB 可以直接基于本地文件系统提供数据库访问服务,一个 Java DB 数据库对应本地文件系统的一个(   )。
   A. 逻辑分区　　B. 目录　　C. 文件　　D. JAR 包
5. 创建数据表之前需要先(   )。
   A. 创建数据库　　B. 插入记录　　C. 查询记录　　D. 删除记录

10.4 开启自己的 Java 探索之旅

经过前面的学习,读者对 Java 应用编程有了比较深入的了解。关于 Java 语言,已经学习的内容如下。

- Java 语言的基本语法,以及如何搭建 Java 编程环境。
- 面向对象程序设计方法,其中包括类与对象编程、类的组合与继承、对象的替换与多态,以及接口编程、泛型编程、异常处理等内容。
- 不同的应用编程场景,以及如何使用 Java API 类库进行应用编程。已学习的应用编程场景包括数值计算、数据集合处理、图形用户界面、数据的输入输出、文字处理、图像和音频处理、多线程并发编程、网络编程和数据库编程等。

在实际应用中,计算机程序还有很多其他的应用场景。例如:

- 程序测试。测试是软件开发过程中的一个重要环节。每次修改程序后都需要对程序进行重新测试,是否可以编写一个测试程序来帮助程序员降低测试工作量呢?
- 媒体播放器。如果需要播放多媒体文件,如何在自己的 Java 程序中编写播放代码,内嵌一个媒体播放器呢?
- 如何编写一个安卓(Android)系统的 App?
- 如何开发一个 Web 网络应用系统?
- ……

学习完本书内容,读者还需要在今后的 Java 学习过程中独自前行,开启自己的 Java 探索之旅。常言道,路要靠自己去走才能越走越宽。本节通过程序测试和媒体播放器这两个应用场景,带领读者体验一下如何根据应用需求去独立探索新的 Java 技术。

10.4.1 单元测试 JUnit

读者在学习程序设计,或者在今后的软件开发工作中一定会碰到**“软件测试”**(software testing)这个术语。不管是出于工作需要或是个人兴趣,如果想深入了解软件测试,该从哪儿开始呢?本书给出的建议是**“从搜索引擎开始”**,例如从百度开始。

1. 发现并追踪“热词”

本书是学习Java语言程序设计的,因此可以将“软件测试”的搜索关键词限定为“Java测试”“Java软件测试”或“如何做Java程序测试”等。通过搜索引擎,您能够查询到很多和Java测试有关的文章。快速浏览这些文章可以大致了解软件测试的基本原理,以及Java软件测试有哪些常用的技术手段。

在学习Java测试的过程中,您会发现很多文章都提到了两个热词:**“单元测试”**(unit testing)和**JUnit**。所谓**热词**,就是大家都在关注,或都在使用的技术。通过Java官网和各种Java论坛,您可以进一步确认:在Java程序开发过程中,很多程序员都会使用JUnit进行单元测试。这说明,单元测试和JUnit值得您花时间去深入了解一下。

对网上查阅到的各种资料进行归纳、整理,可以得到如下一些初步认知。

1) 关于软件测试

单元测试:对单个程序单元进行测试。例如在Java语言中,一个类或一个方法就是一个相对独立的程序单元。Java单元测试,就是单独对一个类或一个方法进行测试。

集成测试:将各程序单元组装起来进行测试。

系统测试:将开发好的程序部署到实际运行环境中进行测试。

……

2) 关于Java单元测试与JUnit

在软件测试中,单元测试是最基础,也是工作量最大的一个环节。JUnit就是为Java单元测试而开发的一个开源类库,它为Java语言提供了一种编写单元测试程序的代码框架。目前,很多实际的Java软件开发项目都使用JUnit进行单元测试。

在Java语言中,一个类包含一组方法,每个方法实现一种算法。在给定输入的情况下,算法应当能够按照设计要求得到对应的输出结果,这个输出结果称作**预期输出**。一个输入,再加上该输入对应的预期输出,就构成算法的一个**测试用例**(test case)。软件测试就是选择一组具有代表性的测试用例,然后检查算法的实际输出与预期输出是否一致。如果一致,则算法正确,否则算法存在错误。

JUnit将每个Java类都作为一个独立的测试单元。单元测试就是测试一个Java类中各方法成员的算法是否正确,以及各方法成员之间是否能协同工作。使用JUnit可以很方便地编写出Java类的单元测试程序。

3) JUnit的使用

JUnit提倡一种“测试与编码同步进行”的编程理念。针对某个已经设计好的Java类,基于JUnit来编写和测试Java类代码的过程如下。

- **编码工程师**编写Java类的定义代码。

- **测试工程师**为 Java 类设计测试用例，然后按照 JUnit 代码框架将测试用例编写成测试代码。测试工程师可以与编码工程师同步开展工作。JUnit 以**断言**(assertion)的形式来比较算法的实际输出与预期输出是否一致。
- **测试工程师**运行测试代码，测试编码工程师所编写的 Java 类。如果出现错误，则提请编码工程师修改。修改后再重新测试，直到没有错误为止。使用 JUnit 编写的测试代码是“一次编写，重复使用”。

在初步了解了单元测试、测试用例和 JUnit 编程理念之后，接下来最重要的事情是试用 JUnit。

2. 试用 JUnit

网上有很多 JUnit 的入门教程，还有一些博客文章通过屏幕截图演示 JUnit 的试用步骤，非常有利于读者模仿学习。通过模仿，读者可以尝试编写出自己的第一个 JUnit 测试程序。**请注意**，在试用环节，读者应尽可能选择最简单的程序例子，并且不要过多地关注技术细节，把程序**走通**是试用环节的第一要务。请读者跟随下面的操作步骤一步一步地模仿进行 JUnit 试用练习。

1) 搭建 JUnit 编程环境

本书所使用的 Java 集成开发环境是 Eclipse 4.7 版。这个版本已经集成了 JUnit 单元测试类库的 JAR 包，因此不需要再单独下载安装 JUnit。

注 1：Eclipse 4.7 集成了三个不同版本的 JUnit 单元测试类库 JAR 包，分别是 JUnit3、JUnit 4 和 JUnit 5。其中，JUnit 5 是最新版本。除了 JUnit，Eclipse 4.7 还集成了很多其他第三方开发的 JAR 包。这些 JAR 包被集中安装在 Eclipse 4.7 安装目录下的 plugins 子目录中，它们被统称为 Eclipse 的“插件”。

注 2：如果读者安装的是其他 Eclipse 版本，或是其他 Java 集成开发环境，请检查其中是否已经包含了 JUnit。如果没有，请单独下载安装 JUnit。JUnit 的官方网址是 http://www.junit.org。

2) 新建 JUnit 试用项目

在 Eclipse 4.7 集成开发环境中新建一个 Java 项目，用于编写 JUnit 试用程序。假设将 JUnit 试用项目命名为 JUnitTest，需要为该项目导入 JUnit 单元测试的 JAR 包。

为 Java 项目导入第三方开发的外部 JAR 包，其导入方法已在 10.3.2 节介绍过。首先选中 Java 项目，再选择 **Project→Properties**，进入项目的属性页对话框，然后在属性页对话框中添加外部 JAR 包，图 10-10 给出了项目 JUnitTest 的属性页对话框。和导入外部 JAR 包不同的是，JUnit 的 JAR 包已经被作为插件集成在了 Eclipse 4.7 的内部，在属性页对话框导入这些插件 JAR 包时应当单击 **Add Library** 按钮，而不是 Add External JARs 按钮。在图 10-10 所示的属性页对话框的左侧首先选中 **Java Build Path**，然后单击对话框界面右侧的 Add Library 按钮，进入下一个“**添加类库**(Add Library)”对话框(见图 10-11)。

先按照图 10-11(a)中对话框的提示选择添加类库 JUnit，然后再按照图 10-11(b)中对话框的提示选择版本 JUnit 5，这样就完成了为项目 JUnitTest 导入 JUnit 单元测试 JAR 包的操作。

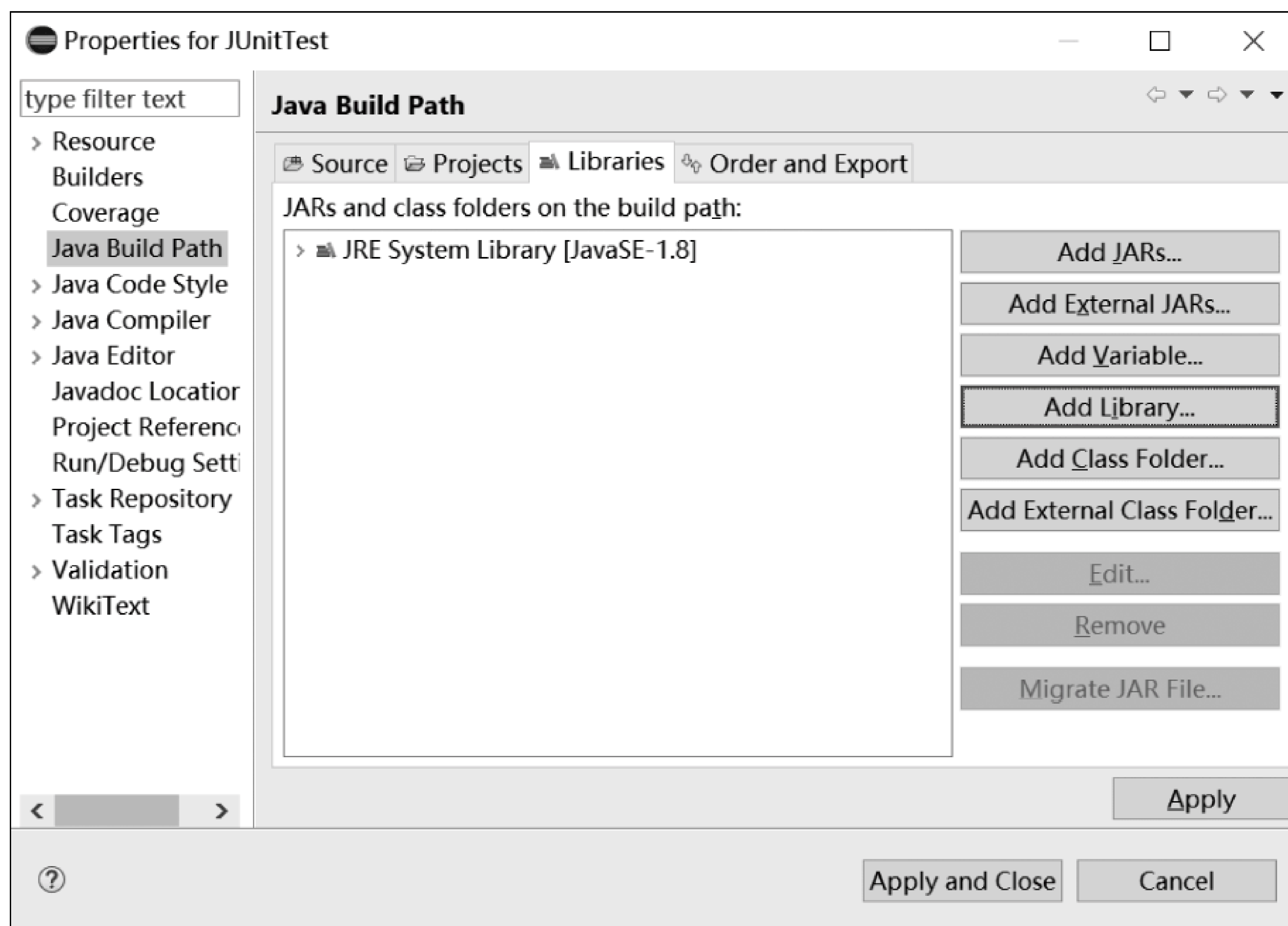


图 10-10 项目 JUnitTest 的属性页对话框

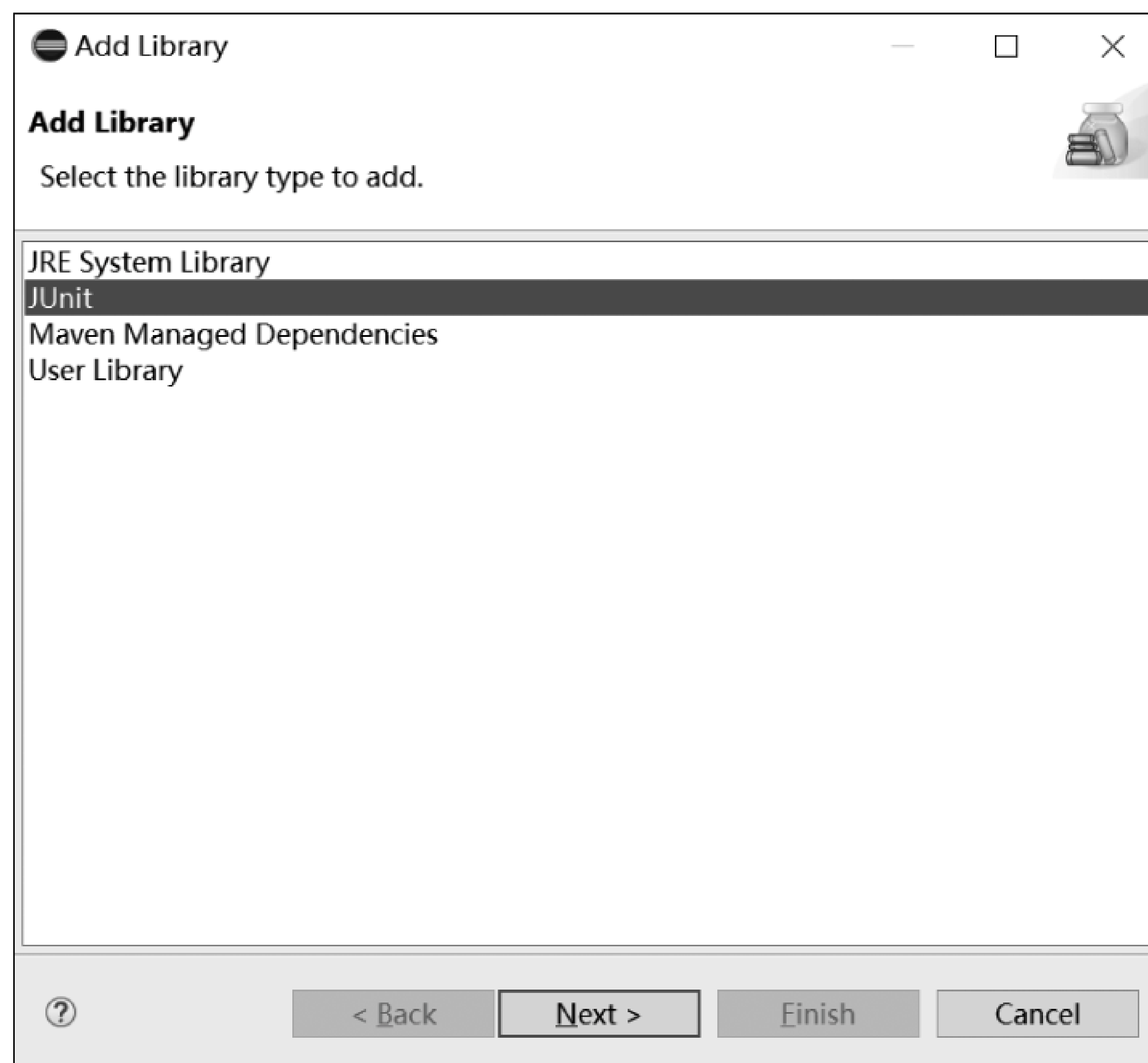
3）在项目 JUnitTest 中新建待测试类

假设需要使用 JUnit 对某个类进行单元测试。首先需要在 JUnit 试用项目 JUnitTest 中新建这个待测试的类，然后编写其类定义代码。在试用 JUnit 时只需要定义一个非常简单的类就可以了。例如，定义一个如下的待测试类 A：

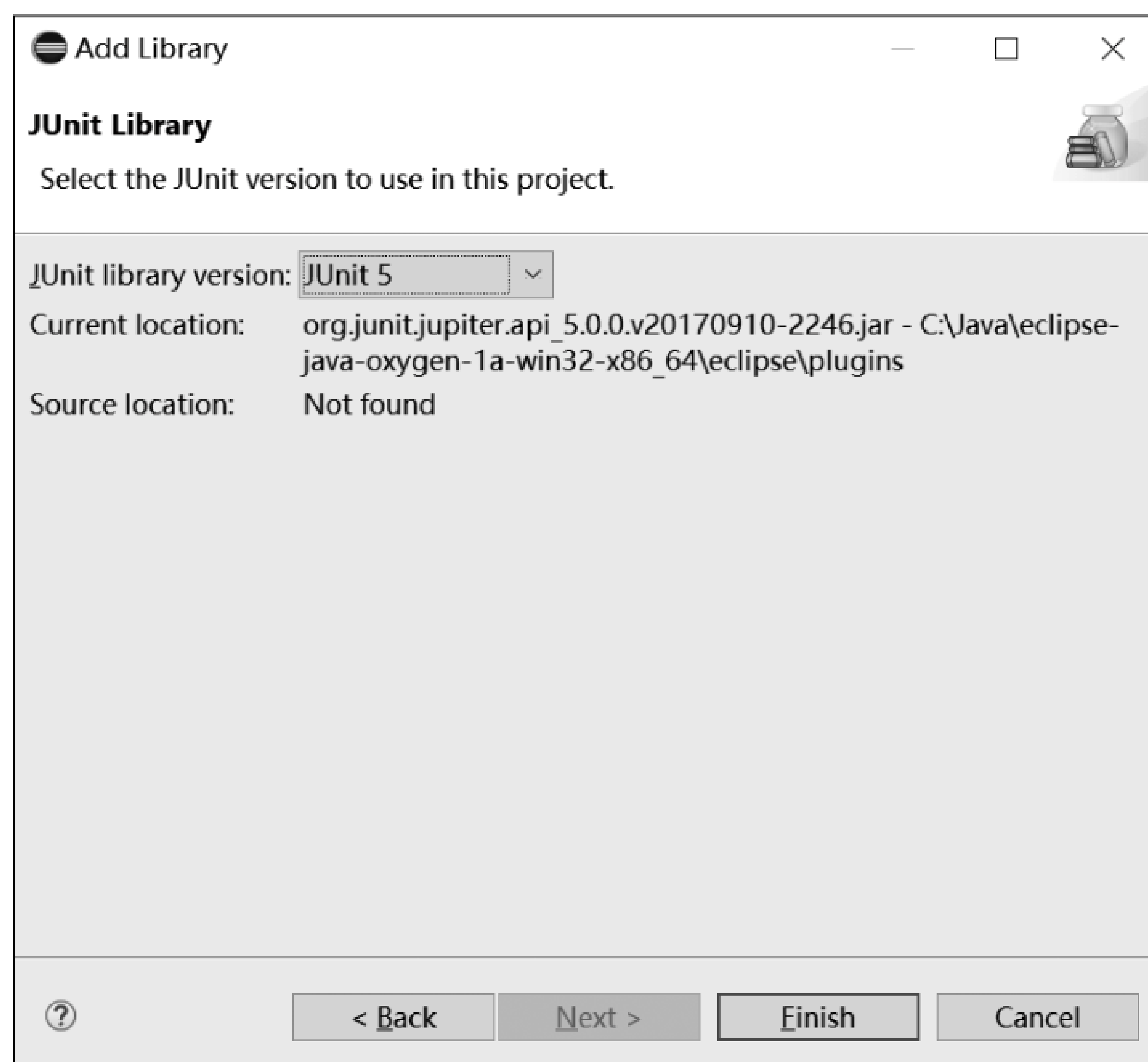
```
public class A {                                        //A 是需要被测试的类
    private int a;                                      //数据成员
    public A(int x) {   a = x;   }                      //构造方法
    public void set(int x) {   a = x;   }               //设置数据
    public int get() {   return a;   }                  //读取数据
    public int getSquare() {   return a * a;   }        //计算平方值
}
```

单元测试就是要测试类 A 中的各个方法成员是否正确，其中包括构造方法 A()、设置数据方法 set()、读取数据方法 get()和计算平方值方法 getSquare()。

单元测试首先要为类 A 中的各方法成员选择测试用例(输入及其预期输出)，然后基于 JUnit 代码框架将测试用例编写成一个测试用例类。例如，基于 JUnit 代码框架来编写一个类 A 的测试用例类，假设将其命名为 JUnitATester。这个测试用例类 JUnitATester 就是待测试类 A 的测试程序。



(a) 选择添加JUnit单元测试类库



(b) 选择JUnit版本

图 10-11　为项目 JUnitTest 导入 JUnit 单元测试 JAR 包

4）在项目 JUnitTest 中新建测试用例类 JUnitATester

选中待测试类 A.java，右击，弹出快捷菜单，选择 **New→JUnit Test Case**，如图 10-12 所示。

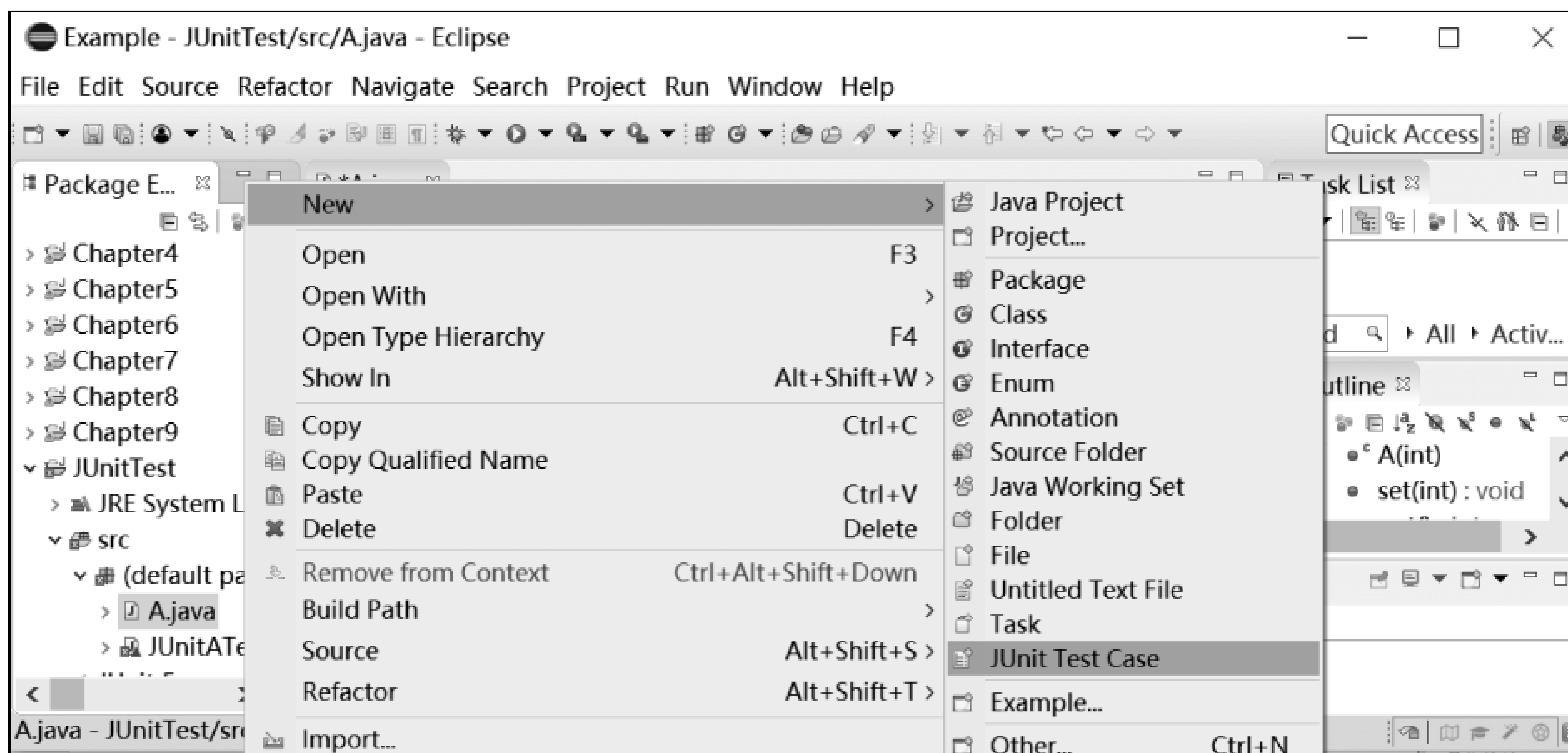


图 10-12　为类 A.java 新建测试用例类的快捷菜单

进入"**新建 JUnit 测试用例**"对话框，如图 10-13 所示。将测试用例类的类名设为 JUnitATester，然后单击 **Finish** 按钮，这样就完成了在项目 JUnitTest 中为类 A.java 新建一个测试用例类 JUnitATester 的操作。

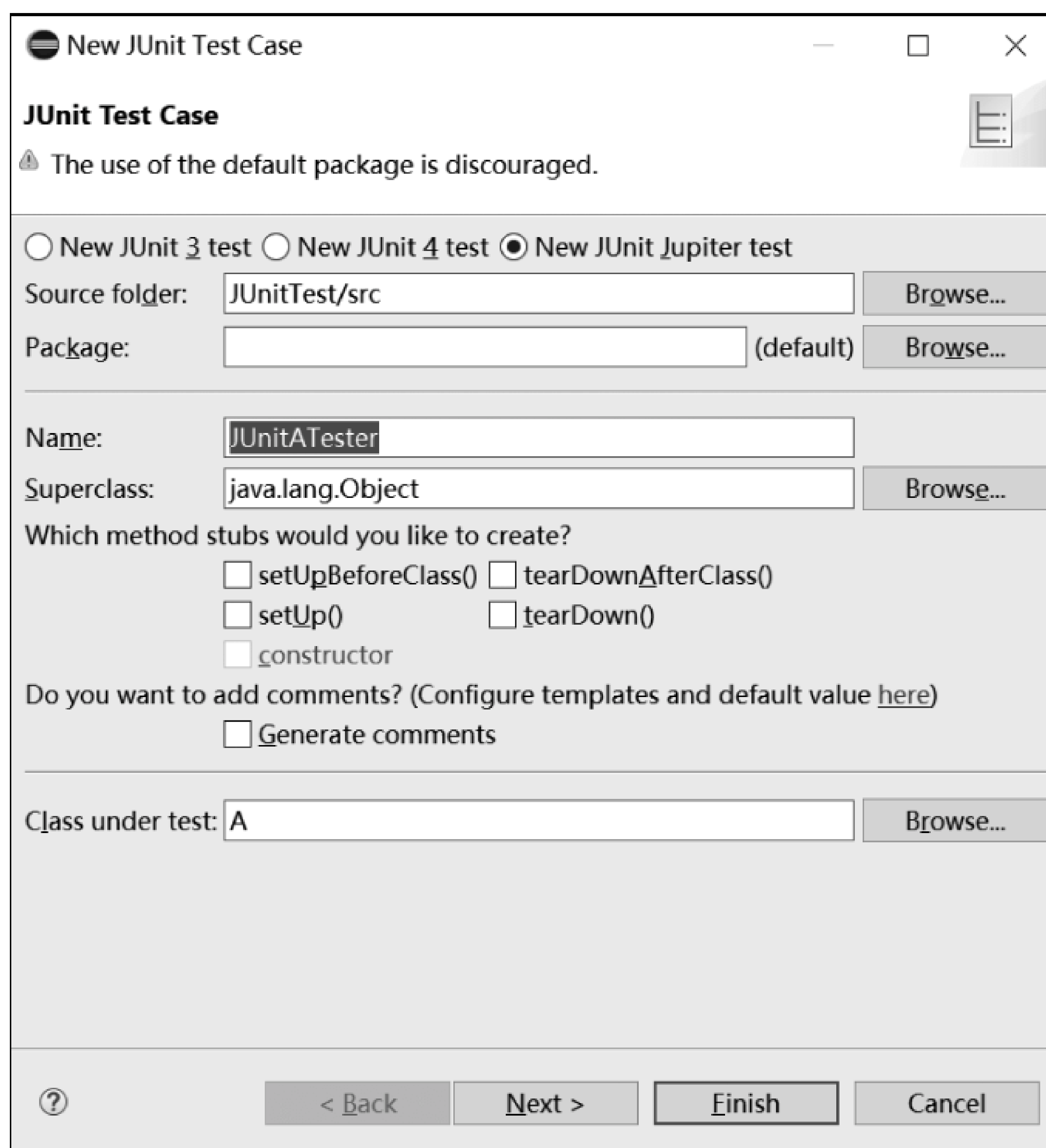


图 10-13　新建 JUnit 测试用例的对话框

Eclipse 会为测试用例类 JUnitATester 自动生成一段如下代码：

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
class JUnitATester {
    @Test
    void test() {
        fail("Not yet implemented");
    }
}
```

下面需要使用 JUnit 提供的**注解@Test** 和**断言 assertEquals()**来编写测试用例类 JUnitATester 中的测试代码。测试代码就是要调用类 A 中的方法成员，然后使用 JUnit 断言来检查其实际输出与预期输出是否一致。例 10-4 给出了一个完整的测试用例类 JUnitATester 的示例代码。

例 10-4　一个完整的测试用例类 JUnitATester 的示例代码(JDBUpdate.java)

```
1   import static org.junit.jupiter.api.Assertions.*;     //导入 JUnit 相关的类库
2   import org.junit.jupiter.api.Test;
3
4   class JUnitATester {                                    //测试用例类
5       /* 测试目的: 检查待测试类 A 中各方法成员的算法是否正确
6        * 测试方法: 使用断言 assertEquals()来检查实际输出与预期输出是否一致
7        */
8       @ Test          //注解: 指定其后面的方法 testA()是一个在测试时需要被执行的方法
9       void testA() {          //测试构造方法
10          A obj = new A(5);   //新建一个对象 obj 并初始化为 5.初值 5 是测试用例的输入
11          assertEquals(obj.get(), 5); //调用 get()方法,使用断言检查实际输出是否为 5
12      }
13      @Test
14      void testSetGet() {             //测试设置数据方法 set()和读取数据方法 get()
15          A obj = new A(0);           //新建一个对象 obj
16          obj.set(6);                 //调用 set()方法,然后再调用 get()方法,检查结果
17          assertEquals(obj.get(), 6); //预期输出为 6,使用断言检查实际输出是否一致
18      }
19      @Test
20      void testGetSquare() {          //测试计算平方值方法 getSquare()
21          A obj = new A( 7);          //新建一个对象 obj,并给一个初值 7
22          assertEquals(obj.getSquare(), 49);
                                        //预期输出 49,使用断言检查实际输出与其是否一致
23  } }
```

请读者根据注释来阅读并理解例 10-4 中的测试代码。

5) 运行测试用例类 JUnitATester

选中测试用例类 JUnitATester.java，右击，弹出快捷菜单。选择 **Run As→JUnit Test**，运行测试用例类 JUnitATester。JUnit 会自动执行其中所有被@Test 注解的方法。这些被@Test 注解的方法会调用类 A 中的方法成员，然后使用 JUnit 断言来检查其实际输出与预

期输出是否一致。Eclipse将测试用例类JUnitATester的测试结果显示在界面左上角的JUnit标签下，如图10-14所示。

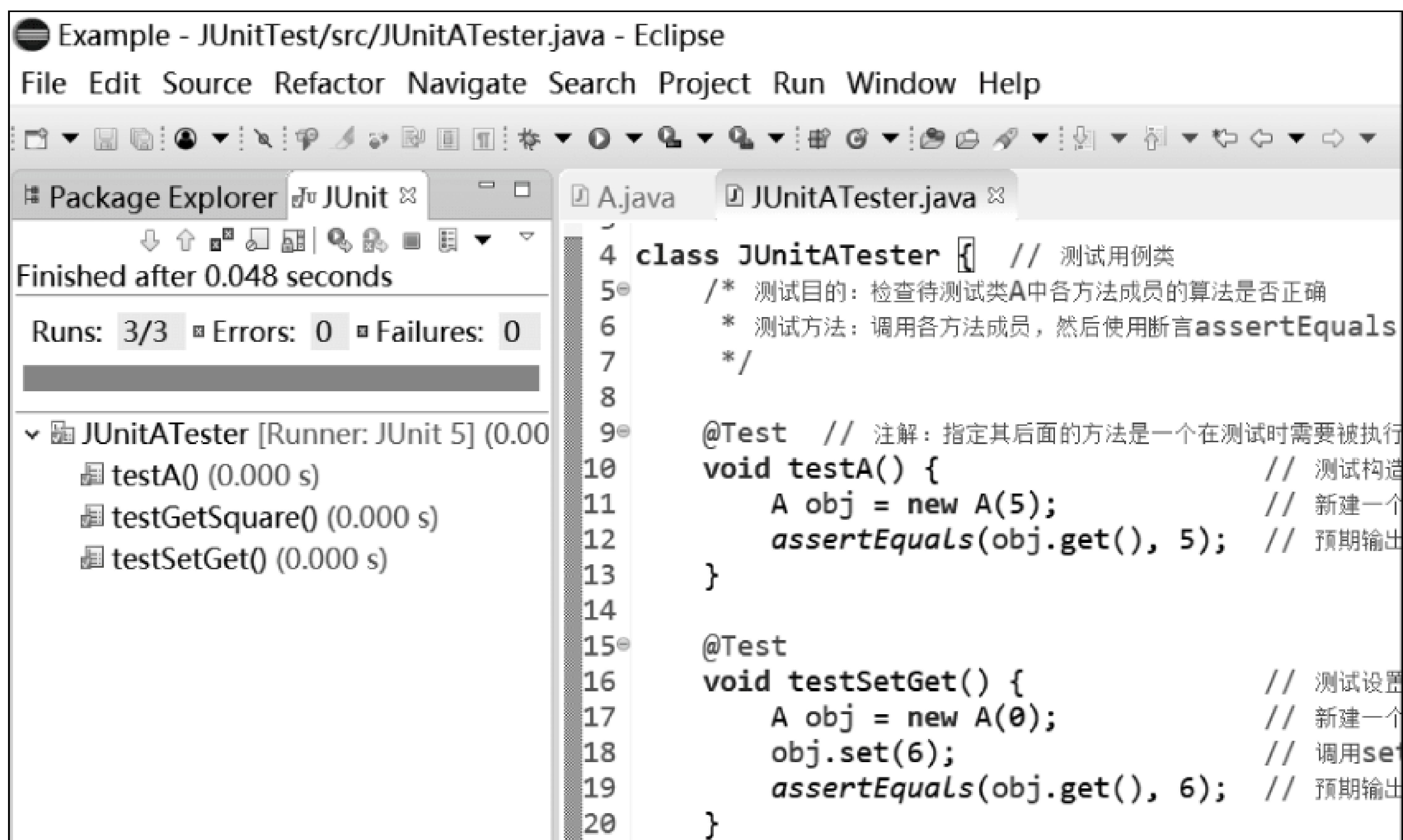


图10-14　运行测试用例类JUnitATester的测试结果(左上角)

在图10-14界面左上角的JUnit标签下可以看到测试用例类JUnitATester的测试结果。其中的"Errors：0"和"Failures：0"表示测试通过，即类A中各方法成员的实际输出与预期输出一致，错误数和失败数为0。

3. 评估JUnit

通过对JUnit的试用，可以初步感受到使用JUnit对Java类进行单元测试还是比较方便的。通常，Java类中各方法成员的签名(或称原型)是在程序设计阶段确定的，而其方法体(即算法代码)则是在编码阶段具体实现的。在编码阶段，方法体中的算法可能会因需求或设计变动而经常修改，每次修改后都需要重新测试。

JUnit实现了Java类测试程序的"一次编写，长期使用"(只要类的接口不变)，这就大大减轻了单元测试的工作量。对JUnit的初步评估结论是：JUnit可以在实际项目中使用。有了这个初步结论，再去仔细阅读JUnit相关的技术文档，并做进一步试用评估。

10.4.2　多媒体框架JMF

如果需要播放多媒体文件，是否可以在自己的Java程序中编写播放代码，内嵌一个媒体播放器呢？换句话说，能否找到一个多媒体类库帮助自己快速编写媒体播放器程序呢？通过搜索引擎，搜索"Java播放器""Java多媒体播放器""如何编写Java多媒体播放器"等关键词，很快就可以发现一个热词——**JMF**。

JMF是**Java多媒体框架**(Java Media Framework)的简称。JMF通过管理器**Manager**、播放器**Player**、控制监听器**ControllerListener**等类或接口为Java程序员提供了一组多媒体编程API。像试用单元测试类库JUnit一样，也需要通过试用来对JMF类库进行评估。

1. 下载安装 JMF

JDK 1.8 安装包中并没有包含 JMF 类库，它需要单独下载安装。JMF 类库可以通过 Java 官网下载，例如下载其 Windows 操作系统安装包 jmf-2_1_1e-windows-i586.exe。

安装 JMF 并在 Java 项目中导入其安装目录下 lib 子目录中的 JAR 包文件 jmf.jar，然后就可以基于 JMF 代码框架编写一个自己的媒体播放器程序了。

2. 模仿编写媒体播放器程序

可以在网上查找一些 JMF 程序的例子，然后通过模仿或直接复制编写一个简单的媒体播放器程序。例 10-5 给出一个基于 JMF 的媒体播放器演示程序。

例 10-5 一个基于 JMF 的媒体播放器演示程序(JMFPlayer.java)

```
1    import java.awt.*;                //导入 java.awt 包中的图形组件类
2    import javax.swing.*;             //导入 javax.swing 包中的 swing 图形组件类
3    import javax.media.*;             //导入 javax.media 包中 JMF 相关的类
4
5    public class JMFPlayer {          //主类：编写一个基于 JMF 框架的媒体播放器程序
6        public static void main(String[] args) {              //主方法
7            JFrame w = new JFrame("多媒体播放器");             //创建程序主窗口
8            w.setSize(600, 400);  w.setVisible(true);
9            w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10           w.validate();
11           //播放一个音频或视频文件(给出文件的 URL)
12           String filename = "file:///D:/samples/1.wav";  //播放一个 WAV 格式的音频文件
13           //String filename = "file:///D:/samples/1.mp4";  //播放一个 MP4 格式的视频文件
14           MMPlayer mmp = new MMPlayer(w, filename);        //创建媒体播放器对象
15   }  }
16
17   //定义一个自己的媒体播放器类，需实现 JMF 的控制监听器接口 ControllerListener
18   class MMPlayer implements ControllerListener {     //媒体播放器类
19       Player p;                                     //Player 是 JMF 的播放器接口
20       JFrame win;                         //程序主窗口：播放器将被作为组件添加到主窗口中
21       String mmFile;                                //等待播放的音频或视频文件 URL
22       MMPlayer(JFrame w, String s) {                //构造媒体播放器对象
23           win = w;    mmFile = s;
24           try {  //处理可能出现的勾选异常
25                   MediaLocator ml = new MediaLocator(mmFile);  //JMF 的类 MediaLocator
26                   p = Manager.createPlayer(ml);     //通过 JMF 的类 Manager 创建播放器对象
27                   p.addControllerListener(this);    //添加控制监听器，监听播放器的事件
28                   p.prefetch();                           //先加载多媒体文件中的数据
29           } catch(Exception e) {  e.printStackTrace();  }
30       }
31       //实现控制监听器接口 ControllerListener 规定的抽象方法 controllerUpdate()
32       public void controllerUpdate(ControllerEvent e) {    //监听并处理播放器相关的事件
```

```
33          if (e instanceof RealizeCompleteEvent) {        //如果是播放器对象创建完成事件
34                  Component mmArea = p.getVisualComponent(); //获取播放区域组件
35                  Component cPanel = p.getVisualComponent(); //获取控制面板组件
36                  //将播放器对象的播放区域组件和控制面板组件添加到程序主窗口中
37                  if (mmArea != null)  win.add(mmArea, BorderLayout.CENTER);
38                  if (cPanel != null)  win.add(cPanel, BorderLayout.SOUTH);
39                  win.validate();
40          }
41          else if (e instanceof PrefetchCompleteEvent) {   //如果是多媒体数据加载完成事件
42                  p.start();                                //开始播放
43          }
44  }  }
```

请读者根据注释来阅读并理解例 10-5 中的媒体播放器程序代码。

3. 运行媒体播放器程序

运行例 10-5 中的媒体播放器程序,测试不同多媒体文件的播放效果。测试结果表明,JMF 能够播放的多媒体文件格式很有限,只能播放 WAV、AVI 等格式的多媒体文件。对于目前流行的 MP3 或 MP4 多媒体文件格式,JMF 都不支持。

试用结果表明,JMF 不支持目前流行的多媒体文件格式。对 JMF 的评估结论是:JMF 不能在实际项目中使用,试用失败。要想编写自己的媒体播放器程序,还得另想办法。

10.4.3 安卓 App 和 Web 网络应用程序

目前,Java 语言在安卓 App 和 Web 网络应用系统这两个开发领域应用最为广泛。在学习完本书内容之后,读者可以进一步学习这两个应用领域的程序开发。

1. 如何编写一个安卓系统的 App

安卓 App 在本质上是一种运行于安卓智能手机上的图形用户界面程序。其编程原理与运行于计算机上的图形用户界面程序基本相同,所不同的是编写运行于安卓智能手机上的应用程序还需要用到一些安卓操作系统所特有的类库。

目前,安卓操作系统由**谷歌**(Google)主导,开发安卓系统 App 主要使用 Java 语言。在自己的计算机上搭建安卓系统 App 开发环境需要:

- 下载安装 **JDK** 和 **Eclipse**(或 Android Studio)。参见本书第 1 章 1.2 节和 1.4 节。
- 下载安装 Android **SDK**(Software Development Kit)和 **ADT**(Android Development Tools)for Eclipse。请参阅安卓系统开发相关的资料或书籍。

2. 如何开发一个 Web 网络应用系统

第 9 章曾介绍过 C/S 架构的网络应用程序,其中包括客户端应用程序和服务器应用程序两部分。C/S 架构最大的不足之处是客户端需要为不同应用系统安装不同的客户端程序,而且每个客户端都要安装。

Web 网络应用系统是另一种架构的网络应用程序,其客户端统一使用浏览器,因此这种网络应用程序架构称作 **Browser/Server** 程序架构,简称 **B/S** 架构。B/S 架构网络应用系

统比较很复杂，所涉及的知识面也很广。除了 Java 语言，还需要掌握 HTTP、HTML、CSS、JavaScript、JQuery、Ajax、JSP、JavaBean、Servlet、Struct＋Spring＋Hibernate（简称 SSH 框架）或 Spring＋SpringMVC＋MyBatis（简称 SSM 框架）等 Web 开发技术。在学习完本书的 Java 基础知识之后，读者可以通过培训机构的 Java 实训课程或网上相关的 MOOC 课程快速学习并掌握这些 Web 开发技术。

最后以一张 Java 网络应用程序架构示意图（见图 10-15）作为本书的结束。

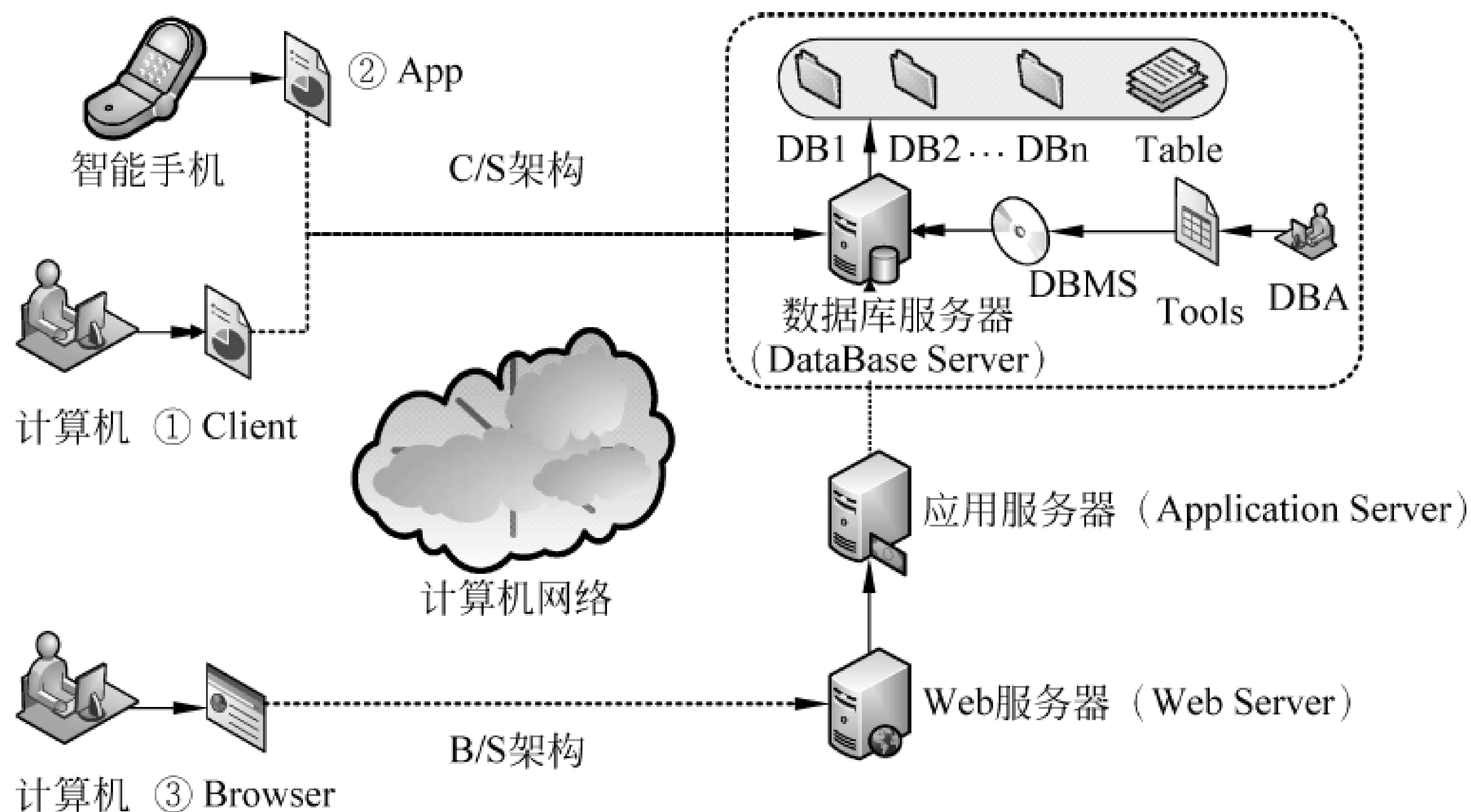


图 10-15 Java 网络应用程序架构示意图

图 10-15 显示了 C/S 和 B/S 两种不同的网络应用程序架构。在 C/S 架构中，不同网络应用需要安装不同的客户端程序，其中运行于智能手机上的客户端程序称作 App。在 B/S 架构中，不管什么网络应用，其客户端程序使用的都是浏览器。操作系统（例如 Windows）一般都自带浏览器，通常不需要再单独安装。

本节习题

1. 下列关于单元测试的描述中，错误的是（　　）。

   A. 一个 Java 类是一个测试单元，其中包含一组方法，每个方法实现一种算法

   B. 在给定输入的情况下，算法应当能够按照设计要求得到一个预期输出

   C. 输入再加上其预期输出就构成了算法的一个测试用例

   D. 单元测试就是测试 Java 类中的字段成员是否正确

2. 下列关于 JUnit 的描述中，错误的是（　　）。

   A. JUnit 将每个 Java 类都作为一个独立的测试单元

   B. 使用 JUnit 可以测试一个 Java 类中各方法成员的算法是否正确

   C. 使用 JUnit 可以很方便地编写出 Java 类的测试代码

   D. 使用 JUnit 可以很方便地编写出 Java 类的定义代码

3. JUnit 通过检查方法的（　　）来测试其算法是否正确。

   A. 源代码　　B. 形参列表　　C. 返回值类型　　D. 返回值

4. 下列编程技术中,安卓 App 通常不会用到(　　)。
   A. 图形用户界面　　B. 网络编程
   C. 数据库编程　　D. JMF 多媒体框架
5. B/S 架构网络应用系统的客户端程序是(　　)。
   A. 浏览器　　B. PDF 阅读器　　C. Word　　D. Excel

本章学习要点

- 了解数据库的基本原理,学习 SQL 和 JDBC 编程框架。
- 熟练运用 JDBC API 编写数据库应用程序。
- 本章所学习的数据库知识已基本能够满足数据库编程的需要。如果希望深入学习数据库,读者可以选修专门的数据库课程,系统学习数据库相关的基础理论和设计方法。
- 开启自己的 Java 探索之旅。

本章习题

编程实验。请读者跟随 10.3 节 JDBC 编程的实验步骤自己动手编写一个完整的 Java 数据库应用程序。



附录A 各章“本节习题”参考答案

第 1 章　认识 Java 语言

1.1　从 C/C++到 Java：ADACD
1.2　Java 开发包 JDK：DCDDA
1.3　Java 程序和 Java 虚拟机：CABAC
1.4　Java 集成开发环境：CDADD

第 2 章　Java 语言基础

2.1　数据类型：CDDCB BAD
2.2　变量与常量：BDDDD CDD
2.3　运算符与表达式：DBBCA CDDCC
2.4　算法结构与控制语句：BACBC CDCBD DDDCD

第 3 章　面向对象程序设计之一

3.1　面向对象程序设计方法：DDDDD
3.2　面向对象程序的设计过程：BDCDB
3.3　类与对象的语法细则：DCDCD DCCDC BCCCD
3.4　数组：DADDB ABC
3.5　Java 程序文件的组织：DDDAD ADD

第 4 章　面向对象程序设计之二

4.1　重用类代码：CCBDD
4.2　类的组合：ADDDD
4.3　类的继承与扩展：DDACC BCB
4.4　对象的替换与多态：DBDCC CCB
4.5　抽象类与接口：CCCCCCCB
4.6　4 种特殊的类定义形式：DBDDD

第 5 章　Java 基础类库

5.1　数学类 Math：DACAC
5.2　字符串类：DACDC
5.3　基本数据类型的包装类：BCCBB
5.4　Java 语言的根类 Object：ABDAD
5.5　系统类 System：DABCD
5.6　异常处理：ABDDC DAB
5.7　泛型与数据集合类：CDDDA BDD
5.8　枚举类型：CCABD
5.9　Java 源程序中的注释和注解：DCBAA

第 6 章　图形用户界面程序

6.1　图形用户界面：DADDD DDC
6.2　编写图形用户界面程序：DACAA ABB
6.3　响应用户操作：AAAAA ABA
6.4　常用图形组件：AAABB ABB
6.5　对话框：BAADD
6.6　鼠标事件和键盘事件：ADDDB
6.7　Java 小应用程序 Applet：BCACD

第 7 章　输入输出流

7.1　Java 输入输出流：DADAD ACDCD
7.2　标准 I/O：BADDA DAD
7.3　文件及文件 I/O：DADBC CDD
7.4　序列化及二进制文件 I/O：CABAD BAA
7.5　文本处理：ACDDD
7.6　图像处理：DBBCA
7.7　声音处理：DDABC

第 8 章　多线程并发编程

8.1　多线程并发程序：ADBAC
8.2　多线程编程及并发调度：ADDBA
8.3　多线程之间的并发与互斥：DBDDC
8.4　多线程之间的协同：ADADB DDC
8.5　定时执行的线程：ABBAB

8.6 swing 框架中的线程：DBCAA

第 9 章 网络编程

9.1 计算机网络的基本原理：DADDB DDC
9.2 网络服务与网络资源：ADDCD
9.3 程序之间的网络通信：CDABB DDB
9.4 基于 UDP 的网络通信：DDAAD BCA

第 10 章 数据库编程

10.1 数据库系统的基本原理：CCABC ADACA
10.2 JDBC 数据库编程代码框架：CAAAA BDA
10.3 JDBC 数据库编程实验：DDDBA
10.4 开启自己的 Java 探索之旅：DDDDA

参 考 文 献

[1] 唐大仕. Java 程序设计[M]. 2 版. 北京：清华大学出版社，2015.

[2] 张思明. Java 语言程序设计[M]. 3 版. 北京：清华大学出版社，2015.

[3] Java 语言官方网站. http://www.oracle.com/technetwork/java/index.html.

[4] 阚道宏. C++语言程序设计(MOOC 版). 2 版. 北京：清华大学出版社，2017.



- 教学目标明确，注重理论与实践的结合
- 教学方法灵活，培养学生自主学习的能力
- 教学内容先进，反映计算机学科的最新发展
- 教学模式完善，提供配套的教学资源解决方案
- 可在清华大学出版社网站下载教学资料



课件下载·样书申请



书圈

清华社官方微信号



扫 我 有 惊 喜



定价：69.00元